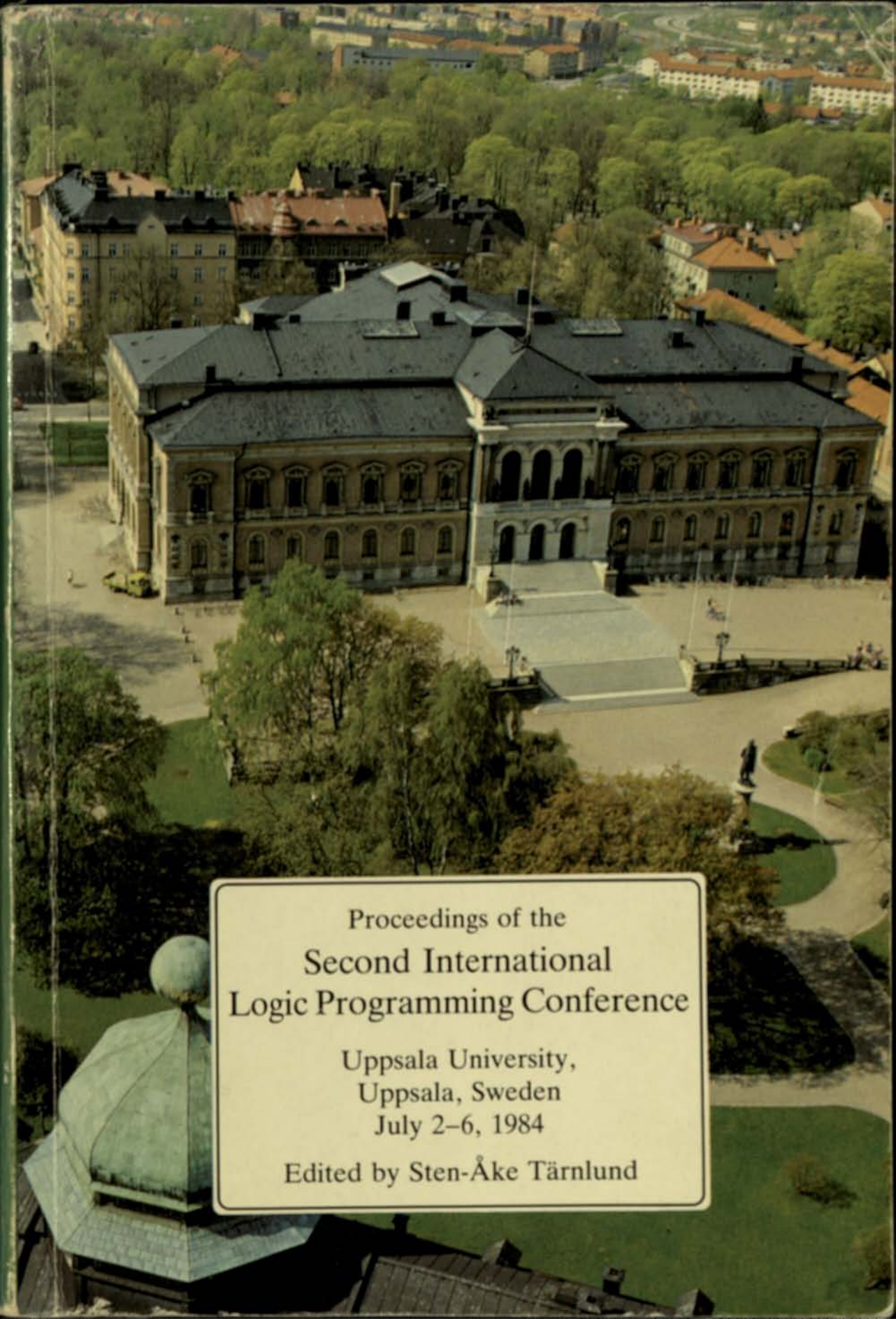


Proceedings of the

Second International Logic Programming Conference

Uppsala University,
Uppsala, Sweden
July 2–6, 1984

Edited by Sten-Åke Tärnlund

Second International
Logic Programming Conference

Ord & Form, Uppsala 1984.

Proceedings of the

Second International Logic Programming Conference

Uppsala University,
Uppsala, Sweden
July 2–6, 1984

Edited by Sten-Åke Tärnlund

CONTENTS

Foreword V
Sten-Åke Tärnlund

Acknowledgement V

Tuesday, July 3, Morning Session

APPLICATION OF LOGIC PROGRAMMING 1
Overall Design of Simpos
S. Tagaki, T. Yokoi, S. Uchida, T. Kurokawa, T. Hattori, T. Chikayama, K. Sakai, J. Tsuji

Prolog as a Tool for Optimizing Prolog Unifiers 13
M. Nilsson

Drawing Trees and their Equations in Prolog 23
J.F. Pique

Tuesday, July 3, Afternoon Session

FOUNDATIONS OF LOGIC PROGRAMS 35
A Logical Reconstruction of Prolog II
M.H. van Emden, J.W. Lloyd

A Comparison of Two Logic Programming Languages: A Case Study 41
M. Szöts

Computation Trees and Transformations of Logic Programs 53
O. Štěpánková, P. Štěpánek

APPLICATION OF LOGIC PROGRAMMING 65
Semantic Interpretation for the Epistle System
M. McCord

On Gapping Grammars 77
V. Dahl, H. Abramson

Wednesday, July 4, Morning Session

LOGIC PROGRAMMING LANGUAGES 89
Eager and Lazy Enumerations in Concurrent Prolog
H. Hirakawa, T. Chikayama, K. Furukawa

Incorporating Mutable Arrays into Logic Programming 101
L-H Eriksson, M. Rayner

Equality, Types, Modules and Generics for Logic Programming 115
J. Goguen, J. Meseguer

LOGIC PROGRAMMING METHODOLOGY

Unfold/Fold Transformation of Logic Programs 127
H. Tamaki, T. Sato

Bounded-Horizon Success-Complete Restriction of Inference Programs 139
M. Sintzoff

An Efficient Bug Location Algorithm 151
D.A. Plaisted

Thursday, July 5, Morning Session

ARCHITECTURE AND HARDWARE FOR LOGIC PROGRAMMING

Or-Parallelism on Applicative Architectures 159
G. Lindstrom

A Class of Architectures for A Prolog Machine 171
L.V. Kale, D.S. Warren

An Architecture for Parallel Logic Languages. 183
J.A. Crammond, C.D.F. Miller

A Highly Parallel Prolog Interpreter Based on The Generalized Data Flow Model . 195
P. Kacsuk

Thursday, July 5, Afternoon Session

APPLICATION OF LOGIC PROGRAMMING

Unification for a Prolog Data Base Machine 207
G. Berger Sabbatel, W. Dang, J.C. Ianeselli, G.T. Nguyen

A Prolog System for the Verification of Concurrent Processes Against
Temporal Logic Specifications 219
P.G. Bosco, G. Giandonato, E. Giovanetti

Logical Levels of Problem Solving 231
Leon Sterling

LOGIC PROGRAMMING METHODOLOGY

Using Symmetry for the Derivation of Logic Programs 243
A-L. Johansson

A Model Theory of Logic Programming Methodology 253
H. Sun, L. Wang

Friday, July 6, Morning Session

FOUNDATIONS OF LOGIC PROGRAMS

A Unified Treatment of Resolution Strategies for Logic Programs 263
D.A. Wolfram, M.J. Maher, J-L Lassez

APPLICATION OF LOGIC PROGRAMMING

FAME: A Prolog Program that Solves Problems in Combinatorics 277
Yoav Shoham

A Mycin-Like Expert System in Prolog 289
A. Littleford

Parlog for Discrete Event Simulation 301
Krysia Broda, Steve Gregory

Friday, July 6, Afternoon Session

LOGIC PROGRAMMING LANGUAGES

Logic Programming by Completion 313
N. Dershowitz, N.A. Josephson

Associative Concurrent Evaluation of Logic Programs 321
K. Nakamura

A Unification Algorithm for Concurrent Prolog 331
J. Levy

ARCHITECTURE AND HARDWARE FOR LOGIC PROGRAMMING

A Memory Management Machine for Prolog Interpreters 343
Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro

AUTHOR INDEX 355

FOREWORD

This is the proceedings of the biennial second international logic programming conference. It gives us a view on the research and progress of Logic Programming and also informs the participants about technical content and details, some of which have to be omitted in the presentations.

Around 100 papers from more than 20 countries and including all continents but Africa were submitted to the conference, they were refereed for their clarity, originality and significance by three members of the program committee.

It is a pleasure to thank all authors who responded to our Call for Papers, unfortunately, restrictions on the size of the conference unabled us to include all fine papers.

I also wish to thank the program committee, special thanks are due to those members who attended the busy meeting in Atlantic City. It is most appropriate to thank Doug DeGroot for his support during this meeting and Alan Robinson and his office for their valuable help.

I should also thank Marianne Ahrne, Elisabeth Askebro and Ingrid Fagerström for their good work in our group for local arrangements.

Finally, I thank our sponsors for their support.

Sten-Åke Tärnlund
Program Chairman

ACKNOWLEDGEMENT

This conference is sponsored by ASEA, Ericsson, IBM, Digital Equipment, Philips and the Swedish ministry of education.

OVERALL DESIGN OF SIMPOS

(Sequential Inference Machine Programming and Operating System)

Shigeyuki Takagi, Toshio Yokoi, Shunichi Uchida, Toshiaki Kurokawa
Takashi Hattori, Takashi Chikayama, Kō Sakai, Junichiro Tsuji

ICOT

(Institute for New Generation Computer Technology)

Mita Kokusai Building, 21F.

4-28, Mita 1-Chome, Minato-ku, Tokyo 108 JAPAN

ABSTRACT

As the first major product of Japanese FGCS (Fifth Generation Computer Systems) project, Personal Sequential Inference Machine (PSI or ψ) is under development. Here we describe the design of the ψ's programming system and operating system SIMPOS, its major language ESP (Extended Self-contained Prolog), and the development tools.

The major research theme of ψ is to develop a logic programming based programming environment including system programs.

The basic design philosophy of SIMPOS is to build a super personal computer with database features and Japanese natural language processing under a uniform framework (logic programming) based system design.

At the end of March 1985, we will be able to show that the logic programming based operating/programming system is working well and has a good human interface.

1 Preface

As the first major product of Japanese FGCS project, ψ is under development. Here we describe the overall design of ψ's Programming System and Operating System called SIMPOS, its major language ESP, and some development tools.

The major ψ research themes are to develop:

- o System programs in logic programming,
- o A programming environment for logic programming.

ψ is the pilot model of the FGCS software development. It is a high-performance personal machine and will be used as the research tool for the FGCS project.

The hardware and firmware design of ψ was completed at ICOT, and the first pilot model has already been manufactured. Its firmware debugging has been finished in March 1984. Installation of SIMPOS was started in February.

SIMPOS has 5 basic design principles. They are:

- o Uniform framework-based system design

  A single uniform PROLOG-like logic programming based framework covers all of the machine architecture, language system, operating system, and programming system.

- o Personal interactive system

  We hope ψ will be one kind of personal and very highly interactive computers similar to many *super personal* computers.

- o Database features

  PROLOG has database facilities that can easily conform to relational database systems. We hope to construct a new programming system and a new operating system that fully uses the database features.

- o Window features

  In order to facilitate high level interaction, ψ uses a bitmapped display and

a pointing device.

- o Japanese language processing

  All computers until now have been based on Western cultures. This is a major disadvantage for peoples of other cultures when they want to use computers. Everyone should be able to use computers in his own tongue. So, the Japanese should be able to use computers in Japanese.

SIMPOS consists of a programming system (PS) and an operating system (OS). OS consists of a **kernel**, a **supervisor**, and **I/O media subsystems**. PS consists of subsystems called **experts**. PS subsystems are controlled by users, but there is a need to coordinate the subsystems or processes. This task is accomplished by the coordinator subsystem.

All the other subsystems are:

Window (OS),
File (OS),
Network (OS),
Debugger/Interpreter (PS),
Editor/Transducer (PS),
Library (PS).

2 ESP

2.1 Overview

SIMPOS is described in a user programming language called ESP. Programs written in ESP are compiled into KLO. KLO is the machine language of ψ and is a PROLOG-like logic based language with several extensions.

As based on a PROLOG-like execution mechanism, ESP naturally has many of the features available in PROLOG. The important ones among them are the use of unification in parameter passing and a tree-search mechanism based on backtracking.

The main features of the ESP language, except for those in common with PROLOG-like languages, are:

- o Objects with states,
- o Object classes and inheritance mechanisms, and
- o Macro expansion.

The assertion and atom name database features (assert, name, etc.) are not directly available, though lower level features (array access, string manipulation, etc.) for implementing them are provided.

2.2 Objects and Classes

The control structure of ESP is basically that of PROLOG: AND-OR tree search by backtracking. However, from another point of view, an ESP program is constructed in an object oriented manner.

An object in ESP represents an axiom set, which is basically the same concept as *worlds* in some PROLOG systems (M. Van Caneghem 1982). The same predicate call may have different semantics when applied in different axiom sets. The axiom set to be used in a call is specified by passing an object as the first argument of a call and prefixing the call with a colon (:).

An object may have time dependent state variables called **object slots**. Values of slots can be examined by certain predicates using their names. In other words, the slot values define a part of the axiom set. The slot values can also be changed by certain predicate calls. This corresponds to altering the axiom set represented by the object. This is similar to **assert** and **retract** of DEC-10 PROLOG, but the way of alteration is limited.

It seemed to be difficult to us, if not impossible, to describe an entire operating system in pure logic without any built-in notion of time dependency. As many of the currently available ideas for the building blocks of an operating system are based on the notion of state, much more investigation is required before starting to write an entire operating system in pure logic (this approach is being tried by Shapiro (E. Shapiro 1984)). This is why object oriented features with side effects are introduced into ESP.

An ESP program consists of one or more class definitions. An object class, or simply a class, defines the characteristics common in a group of *similar* objects, i.e., objects

which differ only in their slot values (only values; slot names are common to the objects belonging to the same class). An object belonging to a class is said to be an instance of that class. A class itself is an object which represents a certain axiom set.

2.3 Inheritance Mechanism

A multiple inheritance mechanism similar to that of the Flavor system (D. Weinreb and D. Moon 1981), rather than the single inheritance seen in Smalltalk-80 (A. Goldberg and D. Robson 1983), is provided in ESP. A class definition can have a nature definition, which defines one or more **super classes**. When a class is a **super class** of another class, all the axioms in the axiom set of the former class are also introduced into the axiom set of the latter class, as well as the original axioms given in the definition of the latter class. By this inheritance mechanism, classes form a network of *is-a* hierarchy.

Some of the super classes and the subclass which inherits them may have axioms for the same predicate name. Since basically the axiom sets of the super classes are simply merged, such axioms are ORed together. Though the order in the ORed axioms has no significance as long as pure logic is concerned, it can be specified in ESP for hand optimization and to control **cuts** and side effects.

Clauses called **demon clauses** define **demon predicates**, which are ANDed, rather than ORed, either before or after, as specified, the disjunction of usual axioms. They are used to add non-monotonic axioms. For example, a door *with a lock* has a **demon** for the predicate **open** for making sure it is already unlocked. In this way, a class **with_a_lock** can be defined separately from the class **door** as a class that contains non-monotonic knowledge.

Part-of hierarchy can also be implemented using the *is-a* hierarchy and object slots. Assume that we want to make instances of class **A** to be a *part of* an instance of class **B**. First, the definition of **A** should be given. Then, a class **with_A** should be defined so that instances of the class **with_A** has a slot which holds an instance of class **with_A**. Finally, class **B** is defined to be a subclass of **with_A**; in other words, the class **B** *is-a* class **with_A**.

2.4 Macros

Macros are for writing meta programs which specify that programs with so and so structures should be translated into such and such programs. Macros can be defined in the form of an ESP program, fully utilizing the pattern matching and logical inference capability of the logic programming language.

In various languages with macro expansion capability, a macro invocation is simply replaced by its expanded form. Though this simple macro expansion mechanism may be powerful enough for LISP-like functional languages, it is never enough for a PROLOG-like logic based language. For example, a macro which expands

$$p(a, f(X+Y))$$

to a sequence

$$add(X,Y,Z), \quad p(a, f(Z))$$

cannot be defined with a simple expansion mechanism.

Macros of ESP are not only expanded at the place of the macro invocation. Certain additional goals can be spliced in before or after the goal in which the macro invocation is given. If the macro is invoked in the head, these goals will be added at the top or the end of the body.

The same macro definition:

$$X+Y => Z \quad \textbf{when} \quad add(X,Y,Z)$$

can be used in two ways. The clause "$add1(M, M+1)$." is expanded into the clause "$add1(M,N)$:-$add(M,1,N)$.", while the body goal "$p(M+1)$" is expanded into a goal sequence "$add(M,1,N), p(M)$".

2.5 Implementation

Currently (in March 1984), a cross compiler of ESP into KLO is available.

The implementation of the object oriented calling mechanism is rather straightforward:

each object has a slot containing a database of codes corresponding to the axiom set associated with the object.

The current implementation uses slot name atoms for accessing object slots. Such access has been found to be very fast thanks to the built-in hashing mechanism of KLO. Certain other firmware supports for accelerating the execution are also planned.

3 Operating System

The operating system part of SIMPOS consists of 3 layers; kernel, supervisor, and I/O media subsystems.

3.1 Kernel

The kernel manages the hardware resources and fills a gap between the ψ hardware and the supervisor. It includes the processor manager which realizes multiple process environments, the memory manager which allocates and deallocates memory space and performs garbage collection, and the I/O device manager which controls the input/output devices.

3.2 Supervisor

The supervisor provides the basic facilities useful for program execution, such as object storages, inter process interactions, and execution environments. For details, refer to (Yokoi and Hattori 1983). Note that a user may extend and modify these facilities as he chooses.

A pool is a container, which is also an object, of objects of any class. A list and an array are examples of pools. An object can be put into or taken from a pool.

A directory is a pool of objects which are associated with a name. An object can be bound and retrieved with a name in a directory. Since a directory can contain another directory as well, a tree of directories is formed, where an object is identified with a pathname.

A stream is a pipe through which objects flow. An object which is put into one end of a stream, will be retrieved at the other end. When no object is in the stream, a retrieve operation is suspended until some object is put into the stream. A stream is used for synchronization and communication between processes.

A channel is defined on the top of a stream to allow message communication between two processes. A port is a message box for two-way communication, connected to other ports. A message sent through the port will arrive at these connected ports, and a message sent from one of these ports will arrive at this port.

A process executes a given program, which is an instance of a program class. The main goal of the program is defined as an instance predicate, and the slots of a program instance hold objects local to the program.

A process has several environments: a program, a library, a world, and a universe. They can be referred to at any point of the program. A world is a sequence of directories held by a process as its working world. A universe is a system directory tree held in a class slot of class **directory**.

3.3 I/O Media Subsystems

I/O media subsystems manage the interfaces with the outer worlds. This subsystem consists of 3 subsystems: window, file, and network.

3.3.1 Window Subsystem

The window subsystem is the main part of high level man-machine interface of ψ (Kurokawa et al. 1984). It supplies multiple logical displays for processes in ψ on a single physical display. The Lisp Machine developed at MIT also supplies such an environment. The Lisp Machine window subsystem manages the major part of the man-machine interface. But our window subsystem manages only the primitive functions. Other functions like echoing are done by other subsystems, transducer, coordinator, etc. This concept increases the modularity of the whole system, and make each subsystem simpler.

For each process, one window is dedi-

cated for its own display and it need not mind other windows. In the window subsystem, each window is defined as an instance of the window class and each predicate for the window is written as methods of the class. So the window manager need not consider the interaction among the windows and each process can consider its window as its own display. Each window is a rectangular area which is a part of the physical screen, and is the communication channel to the process.

In the window subsystem, windows construct a hierarchy. The most superior window is the logical screen, and normal windows (editor window, etc.) are inferior windows of the logical screen. Each window may have inferior windows (called subwindows) within it, and each inferior window can have its own inferior windows. For example, an editor window has command sub-window, text sub-window, etc. Subwindows can neither have a size that exceeds their superior window's size, nor go out from the superior window. They must be inside of the superior window.

Each window may have one of the following 5 states:

selected: Connected to the keyboard. Only one window can have this status at a time.

shown: Completely displayed on the physical screen, and the mouse button click in this window is interpreted using the key-command definition of this window.

exposed: Completely displayed on its superior window. However, when the superior window does not have the shown status, even if the window is completely displayed on the screen, it does not have shown status, but exposed status.

overlapped: Partly or completely hidden by its superior window. This window is hidden by another inferior window of its superior window.

deactivated: Not managed by the window subsystem. Windows in this status will never be shown on the physical screen until activated. However, its memory image is not destroyed.

Table 3–1. Window Status

Status	KB	mouse	output
Selected	done	done	done
Shown	wait	done	done
Exposed	wait	wait	wait
Deexposed	wait	wait	wait
Deactivated	fail	fail	fail

These states are managed by the window manager. The I/O function is determined by these states. The relation between the window states and the I/O functions are shown in table 3-1.

Whenever there is a keyboard input, the window subsystem has to decide which window the input should be sent to. The window manager has the instance slot **selected_window** which keeps the **selected** window. As another input device, ψ has a pointing device **mouse**. The mouse can move anywhere on the display screen, and the window manager can recognize the window, which the mouse click is sent to, by the position of the mouse. The mouse click is treated by the window's definition in only the **shown** window. It is because if the mouse click changes the window's output image, the user may not see it since he cannot see the whole of the not **shown** window, and the window manager cannot recognize which part of the window is hidden.

3.3.2 File Subsystem

The file subsystem provides permanent storage both for data and objects.

A permanent storage of data (records) is a file. Three types of files are available; binary files, table (fixed length record) files, and heap (variable length record) files. A record is identified with its stored position and/or its associated key through an index file. A binder mechanism will be supported so that a user can define a virtual file with many data and index files. A relational database management may be built on these facilities.

A permanent storage of objects is an instance file. It is the main feature of the file subsystem not provided by other machines' ordinary file systems.

A directory file is a file which associates an instance record with a name. A permanent directory is a directory which has a directory file as its permanent storage. When included in a permanent directory, a permanent object is stored as an instance record in an instance file and included in the directory file with a pathname. Therefore, it can be restored even after the system is rebooted.

3.3.3 Network Subsystem

The network subsystem provides three types of interfaces to communicate with other machines.

Inter-machine communication is supported to connect one ψ with another ψ or other different machines. The network subsystem defines the classes **node**, **socket**, **cable**, and **plug** to implement the communication.

Inter-process communication allows two processes on different ψ nodes to communicate with each other, just as if they exist on the same node. A remote channel is defined to represent an original channel on the other node. A process can send a message to the remote channel and another process on the remote node can receive it from the corresponding original channel.

The introduction of remote objects is a main feature of the network subsystem. A remote object represents an object in a remote node. It can be manipulated just as an ordinary object.

4 Programming System

The programming system of SIMPOS is a collection of expert processes. An expert process is a process which has an independent communication window (called **e_window**) with the user. It performs the special action upon the user's request.

This view is different from the views such that the programming system is a collection of dumb software tools, nor is it a collection of programs to support the program production. Our view frees us from the overhead of the controlling process to manage the available tools or the information between the programs.

From the user's viewpoint, he can invoke, control, and terminate any expert through the expert's **e_window**. He need not navigate the complicated process invocation tree to accomplish his task. He need not bother about the unexpected destruction of his work through wrong navigation.

4.1 Coordinator

In SIMPOS, there is no explicit supervising process such as Shell in Unix. However, there is a work-behind process named **Coordinator**. **Coordinator** itself is not an expert process but a process that manages the set of experts.

As noted above, the user may think that he controls the expert directly through the window, but actually, **coordinator** helps the user's control via the window interface that is the associated key command table of the window.

The principal functions of **coordinator** are as follows:

- Send a user's key command through the window to an expert,
- Create, delete, and activate an expert via system_menu,
- Get and process special commands from an expert, and
- Help communications between experts via the whiteboard.

The whiteboard is just like a blackboard where an expert puts a message to another expert, who in turn picks up the message by the user's instruction.

The other way to solve this communication problem is to set a communication channel with another expert. But, in this case, the channel should be set between the experts before the user decides the partner of the expert. It is not easy to tell

who talks to who before communication becomes necessary.

The ultimate solution in this line would be to set a communication channel between any two experts, even though the cost becomes very high as the number of experts grows. And still, a few problems remain. The user may change the partner after he ordered the expert to put the message. It may difficult to denote both the partner and the message using only the mouse click.

Using the whiteboard, we can set virtually complete communication channels between experts. The user can select any expert after he has ordered one to put the message. This operation will be realized with one mouse click.

Each user has a directory to create experts. It contains the experts' names and the program names to create experts. The user can change the directory and the command table as he likes.

A user has his own directory which is inherited from the system's common directory, i.e., the standard set of experts.

An expert has its own set of key command table associated with its window. However, **Coordinator** permits the user to change the key command table of the window only when that window accepts the *change key command table* command from the user.

This freedom is achieved at the least cost of execution. This minimum overhead and the maximum provision of user control is the main theme of **Coordinator**.

4.2 Debugger/Interpreter

This subsystem interprets programs and provides information concerning the control flow of the programs. The basic facilities of the Debugger/Interpreter subsystem is similar to the debugging facility of DEC-10 PROLOG (D. L. Bowen et al. 1981). New features are:

- o Procedure and clause box control flow model,
- o Calls between interpretive and compiled codes, and
- o Multi-window user interface.

DEC-10 PROLOG uses *Box Control Flow Model* for its debugger. It considers that each predicate is the debugging unit. In this view, each clause looks like a black-box and cannot be traced whether the unification of its head or body fails. The predicate call simply fails in both cases. However, it is often the case that the clause head is correctly selected, but the definition of the body is erroneous. When the *Procedure and Clause Box Control Flow Model* is used, it is possible to check whether unification of the head or that of the body fails (see fig. 4-1).

In ψ, it is possible for interpretive and compiled codes to mutually call each other. However, Debugger cannot trace in the compiled code. Debugger treats the invocation of compiled codes just like a simple built-in predicate invocation. If interpretive codes are invoked from compiled codes,

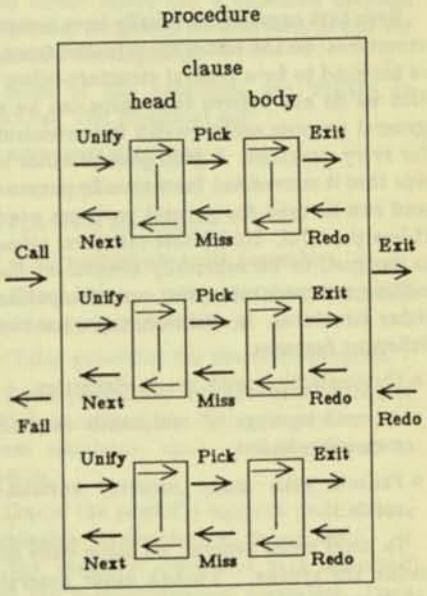


Fig. 4-1. Procedure and Clause Box Control Flow Model for interpretive code

there is no way to pass the trace information to the interpretive codes. In such a case, Debugger restarts tracing with no trace information.

ψ has a bitmapped display screen. Debugger uses the window subsystem that offers a multi-window user interface with the mouse. A user can select one of the control options at break points, look at ancestors or spy points, check the values of slots, or see the class definitions using the library subsystem. This information is shown in sub-windows of Debugger and all the selections can be done using the mouse click.

4.3 Editor

An editor is a typical component of a programming system and an indispensable software tool in using a computer system. Though there can be editors to manipulate abstract structures completely different from texts, here we limit our discussion to the editors which edit texts or data expressed in texts.

Even text expressions usually have nested structures. So the editor for ψ (called Edips) is designed to be a general structure-editor. But we do not believe that there can be a general purpose editor which is convenient for every structure. A good general editor is one that is convenient for a specific purpose and can be used for general purposes even if less powerful. Under this criterion, Edips is designed to be especially convenient for editing ESP programs and can manipulate other structures. In addition, Edips has the following features:

- Customization with macro definition,
- A small number of commands easy to memorize, and
- Failsoft with many recovery environments.

To make Edips general, we allow users to define the syntax. Though other general structure-editors usually use BNF, we do not adopt it because usual editing operations are neither to trim a branch of the syntax tree nor to traverse the tree. Editing operations are more closely related to the text expression of edited data. So we adopted an operator precedence grammar with user definable parentheses. An operator precedence grammar is more simple and has better correspondence to the text expression.

Every token in the text expression of edited data is classified into 6 categories:

- Atom,
- Prefix operator,
- Infix operator,
- Postfix operator,
- Left parenthesis, and
- Right parenthesis.

Each operator has a precedence. For editing purpose, however, too many precedence levels should not be adopted, because precedence introduces structures without direct correspondence to the text structure. As for an ESP editor, 2 or 3 levels are necessary and sufficient. They are for:

- logical symbols such as

  ":-", ";", ","

- function symbols such as

  "+", "-", "*", "/".

If necessary,

- predicate symbols such as

  "<", ">", "="

will be added.

As explained above, the operator precedence grammar is very simple, but has enough expressive power to define the syntax of almost all structured programming languages.

It is desirable that the parser and the pretty printer for the grammar can be used by other programming tools such as compiler, interpreter and debugger. So, those tools are made as separate utilities from the editor. Edips consists of the editor kernel and those utilities which are also used by other tools.

4.4 Library

The library subsystem manages all the classes and predicates on ψ. It controls the registration of classes, loading program files, compiling, and building class objects by the analysis of inheritance.

Each class has a class source file, a class template file, and a class object file on some secondary storage. Class templates and class objects exist only in the main storage, but are saved to and restored from the secondary storage.

Class source files are text files coded by the users. A class source file can have just one class definition. Like source files, template files and object files also have just one class information in each.

A class template is built from a single source file. It holds all the information of that class except those from inheritance analysis. The predicates of that class are kept as interpretive codes when the template is built. They are compiled when the user requests. After the compilation, both interpretive and compiled codes are kept. Templates can be saved or restored before compiling the predicates.

Class objects are built from some class templates. In a class object, all the inheritances are analyzed and solved. It is an executable image of an object oriented program.

Another feature of the library subsystem is to manage predicates. It contains the features of referring to one predicate of a class, i.e., object oriented invocation, and the invocation from compiled codes to interpretive codes or the converse. This mechanism is implemented by indirect references. All the invocation of predicates are done via indirect references. When some interpretive codes are invoked, that indirect word points the entry of the interpreter. This mechanism causes a uniform invocation scheme even if both the interpretive and compiled codes are mixed.

For object oriented invocation, it is necessary to find which method should be invoked during the execution time. Here, the library has to distinguish those predicates that have the same predicate name but are defined in different classes. In the compiled codes, all the references are processed and changed to the direct invocation of the specific predicate, but in the interpretive codes, the library has to search the predicates during the execution time.

The compiler is simply a subroutine of the library subsystem. It compiles a single predicate from interpretive codes. This process is done only in main storage. After the compilation, library holds both interpretive and compiled codes. The user can specify which code should be used for building up a new class object. The template file is automatically rebuilt after the compilation.

5 Development Tools

Almost all of the OS/PS programs are written in ESP. Since they were designed and coded *before* the ψ machine becomes available, we need a cross system of ESP for software development.

Most of the programs are written in PROLOG. The programs are:

- o ESP cross simulator,
- o KLO cross compiler,
- o KLO cross assembler,
- o ψ microprogram cross assembler,
- o Cross linkage editor for both KLO and microprogram,
- o Table generator for the microprogram.

Some programs, the execution speed of which is critical for debugging (microprogram simulator, etc.) are developed in PASCAL.

One of the powerful support tools is **Customizable Assembler** (S. Takagi 1983). It is the kernel of both the KLO assembler and the ψ microprogram assembler. Only the machine-dependent parts such as the length of the object word, field definitions, mnemonic definitions, and checking conditions are changed. Machine-dependent

parts are pre-processed and are compiled with the assembler kernel into a machine-dependent assembler.

The definition of KLO is about 500 lines while the definition of the ψ microprogram is approximately 1100 lines. About 80% of them are conversion tables from mnemonics to field values. The kernel part is about 900 lines of PROLOG program. Compared with many so-called *generalized assemblers* or *universal assemblers*, this assembler has only 1/5 to 1/10 as many codes. Its assembly speed is, however, approximately comparable.

Using PROLOG's unification and backtracking mechanism, it is possible to write a sophisticated error-checking routine. If one field overlaps another, the unification fails and the next alternative value setting is tried. Setup conditions are processed in the same way. If an assembler variable X is unified to the value **case_1** while one field is processed, the process for any other field cannot unify **case_2** for X. So, the unification fails and the process backtracks. Finally, when all of the unification is successfully completed, the object bit-pattern is generated and written out to the object file.

6 Conclusion

A logic programming based inference machine (ψ) and its Programming/Operating System (SIMPOS) is now under development. The first pilot hardware has already been manufactured and firmware debugging was finished. Installation of SIMPOS was started in February.

The first release of ψ and SIMPOS for FGCS research and development will be at the end of March 1985. We will continue its improvements and enhancements. At that time we will be able to show that the logic programming based Operating/Programming system is working well and has a good human interface.

Many investigations and researches are necessary for building logic programming based programming and operating systems. We hope this work will contribute to such researches.

ACKNOWLEDGMENTS

The authors thank Mr. G. Hagio and Mr. H. Ishibashi for their contribution to our project.

APPENDIX I

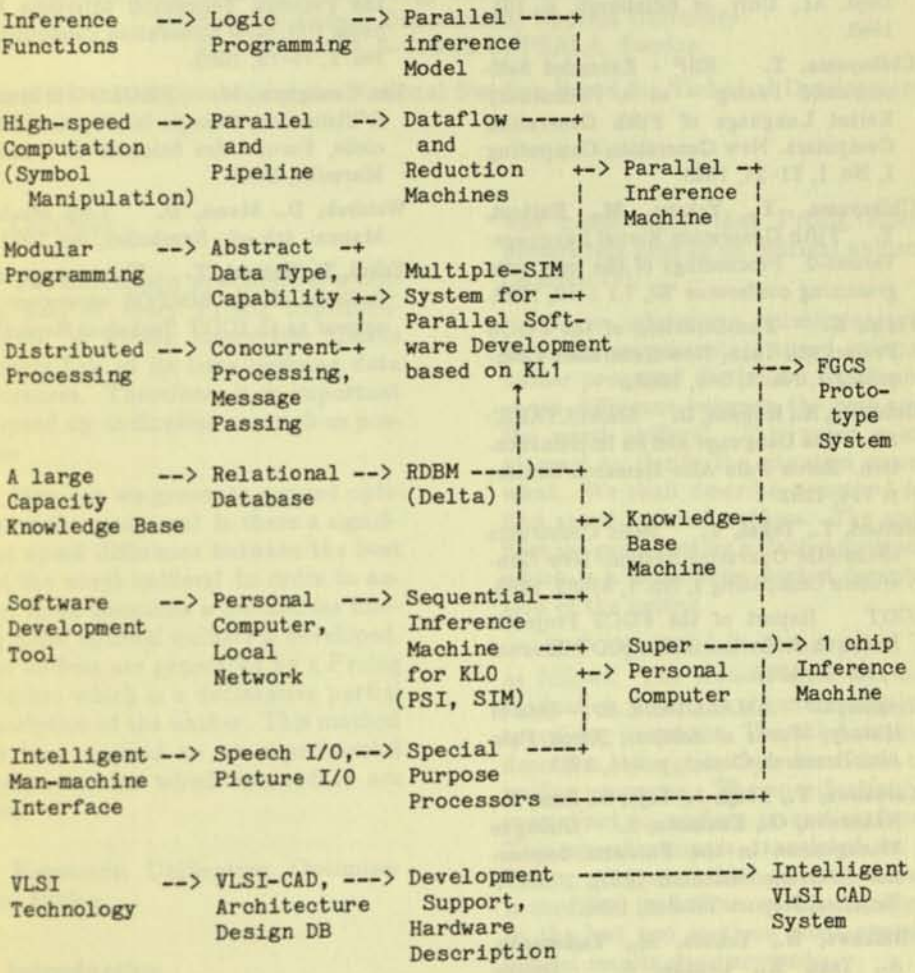


Fig. I-1 An Approach to the Fifth Generation Computer

REFERENCES

Bowen, D. L., Byrd, L., Pereira, F. C. N., Pereira, L. M., Warren, D. H. D. DECsystem-10 PROLOG User's Manual. Dept. AI., Univ. of Edinburgh, p. 101, 1983.

Chikayama, T. ESP – Extended Self-contained Prolog – as a Preliminary Kernel Language of Fifth Generation Computers. New Generation Computing 1, No. 1, 11–24, 1983.

Chikayama, T., Yokota, M., Hattori, T. Fifth Generation Kernel Language: Version-0. Proceedings of the logic programming conference '83, 7.1 1–10, 1983.

Fuchi, K. The Direction of the FGCS Project will Take. New Generation Computing 1, No. 1, 3–9, 1983.

Goldberg, A., Robson, D. SMALLTALK-80 — The Language and its Implementation. Xerox Palo Alto Research Center, p. 714, 1983.

Hattori, T., Yokoi, T. Basic Constructs of the SIM Operating System. New Generation Computing 1, No. 1, 81–85, 1983.

ICOT Report of the FGCS Project's Research Activities 1982. ICOT Journal 1, No. 2 , 1983.

Krasner, G. SMALLTALK-80 — Bits of History, Words of Advice. Xerox Palo Alto Research Center, p. 344, 1983.

Kurokawa, T., Tsuji, J., Tojo, S., Ilma, Y., Nakazawa, O., Enomoto, S. Dialogue Management in the Personal Sequential Inference Machine (PSI). ICOT Technical Report TR-046, 1984.

Nishikawa, H., Yokota, M., Yamamoto, A., Taki, K., Uchida, S. Design Philosophy and Architecture of the Sequential Inference Machine PSI (In Japanese). Proceedings of the logic programming conference '83, 7.2 1–12, 1983.

Shapiro, E. Systems Programming in Concurrent Prolog. Eleventh Annual Symposium on Principles of Programming Languages (to appear).

Takagi, S. Customizable microprogram assembler. ICOT Technical Report TR-021 (In Japanese), p. 25, 1983.

Uchida, S., Yokota, M., Yamamoto, A., Taki, K., Nishikawa, H. Outline of the Personal Sequential Inference Machine PSI. New Generation Computing 1 No. 1, 75–79, 1983.

Van Caneghem, M. PROLOG II Manuel D'Utilisation, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, 1982.

Weinreb, D., Moon, D. Lisp Machine Manual, 4th ed., Symbolics, Inc. 1981.

Yokoi, T., Hattori, T. The concepts and facilities of the SIMPOS supervisor (to appear as an ICOT Technical Report).

Prolog as a Tool for Optimizing Prolog Unifiers

Martin Nilsson

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Department, Uppsala University
P.O. Box 2059, S–750 02 UPPSALA, Sweden

This work was supported by the National Swedish Board for Technical Development (STU).

1. Abstract

The unification procedure is a central part of every Prolog implementation. A Prolog interpreter spends roughly half of its time unifying data structures. Therefore, it is important to speed up unification as much as possible.

How can we generate a speed optimal unifier program? Is there a significant speed difference between the best and the worst unifiers? In order to answer these questions a method for finding speed optimal unifiers is developed. The unifiers are generated by a Prolog program which is a declarative partial description of the unifier. This method has been applied to an experimental interpreter, for which some data are given.

Keywords: Unification, Optimization, Prolog.

2. Introduction

The derivation of efficient unification algorithms from specifications has been studied by a number of researchers, e.g. (Eriksson 83).

However, it seems that few people have studied the problems of finding the most efficient *implementations* of unification algorithms, although it was noted early that Prolog interpreters rely heavily on efficient unifiers (Warren 77).

Some interesting questions are: How can we generate a speed optimal unifier program? Is there a significant speed difference between the best and the worst unifiers? This paper is an attempt to clarify the situation somewhat. We shall describe a method to find speed optimal unifiers. The unifiers are generated by a Prolog program which is a declarative partial description of the unifier.

The organization of this paper is as follows: The section after this introduction describes a general way of specifying programs. The third section describes some primitives for a class of unifier programs. The specification is specialized to unifiers in section four. This specification is translated to a Prolog program, which in section five is modified to find an optimal unifier. In the last two sections some experimental results are discussed.

3. A class of programs

Two different languages are used in this article. One is the *specification language*, which is First Order Logic. The specified unifier programs are written in a second language, the *programming language*. Programs in this lan-

guage are *ground terms* in the specification language.

First, we shall describe how programs in the second language can be specified. Henceforth, if we talk about *programs*, and the language is implicit, we mean programs in this second language.

A program can be seen as a (possibly degenerate) binary tree. An execution follows a path from the root node to a leaf. At every node during execution, the program has a *state*, on which some primitive *operation* is performed. The state is held in a set of state variables, implemented as memory cells in the physical program.

A program can be constructed from three kinds of operations: *tests*, *transformations*, and *terminals*. A *test* node has two successor nodes. It does not change the state variables, but merely directs execution to either the right or the left successor depending on the current state. A *transformation* has only one successor, but may change the state. A *terminal* has no successor. It is responsible for returning some output from the entire program.

Syntactically, programs are linearized as ground terms. A program can be a terminal. Another possibility is $SEQUENCE(x, y)$, where x is a transformation and y is a program. The meaning of this is that x is executed before y. A program can also be $IF(x, y, z)$, where x is a test and y is a program (the "then-branch") and z is a program (the "else-branch").

In other words, p is a *program* with respect to a set of possible input states si iff p satisfies $program(si, p)$, where *program* is defined as

$$\begin{aligned}
&\forall si, p\ (program(si, p) \leftrightarrow \\
&\quad correctstateset(si) \wedge \exists op, p_1, p_2, so_1, so_2(\\
&\qquad terminal(si, p) \\
&\qquad \vee\ p = SEQUENCE(op, p_1) \\
&\qquad\quad \wedge\ transformation(si, op, so) \\
&\qquad\quad \wedge\ program(so, p_1) \\
&\qquad \vee\ p = IF(op, p_1, p_2) \\
&\qquad\quad \wedge\ test(si, op, so_1, so_2) \\
&\qquad\quad \wedge\ program(so_1, p_1) \\
&\qquad\quad \wedge\ program(so_2, p_2)))
\end{aligned}$$

The so variables are output sets of states, i.e. states that come out of operations. The predicates *terminal*, *transformation*, and *test* are supposed to be false whenever their second argument is not an operation name.

Suppose that *TRM* denotes a typical terminal operation in our program. *TRM* could be specified by

$$\begin{aligned}
&\forall si, so\ (terminal(si, TRM) \leftrightarrow \\
&\quad precondition(si, TRM))
\end{aligned}$$

if the set of possible input states si is specified by the precondition.

Likewise, suppose that *TRNS* is a transformation operation. The relation between si and the set of corresponding output states so is specified by a postcondition:

$$\begin{aligned}
&\forall si, so\ (transformation(si, TRNS, so) \leftrightarrow \\
&\quad precondition(si, TRNS) \\
&\quad \wedge\ postcondition(si, so, TRNS))
\end{aligned}$$

A typical test operation, say *TST*, tests if the input state satisfies some condition:

$$\begin{aligned}
&\forall si, so_1, so_2\ (test(si, TST, so_1, so_2) \leftrightarrow \\
&\quad precondition(si, TST) \\
&\quad \wedge\ postcondition_1(si, so_1, TST) \\
&\quad \wedge\ postcondition_2(si, so_2, TST))
\end{aligned}$$

The output set of states so_1 would be the subset of si where the condition is satisfied, while so_2 would be the subset where it is not.

Every primitive operation requires satisfaction of some precondition on the set of possible input states. The precondition on the input checks that the operation is always applicable. There are also postconditions on the output states. A postcondition specifies the set of output states for a given set of input states. Postconditions for test operations should also check that the operation is nontrivial: Both sets of output states should be nonempty.

Some observations can be made here: No program can generate a run-time error, since the applicability of each operation is guaranteed by the operation's precondition. Every program can be insured to be partially correct by having a correctness criterion as a precondition of every terminal. All programs will also be totally correct, if it can be proved that every primitive operation requires a finite amount of execution time.

4. Unifier building blocks

Since we are going to specialize our programs to unifier programs, we shall describe primitive operations for a reasonable class of unifiers. Although we shall have to make some assumptions about implementational details, the principles should apply to other kinds of unifiers as well.

The unifier is assumed to operate on data structures which are Prolog terms represented as binary trees. This is how terms are implemented in, for instance, FOOLOG (Nilsson 83) and HORNE (Frisch, Allen, Giuliano 83/84). A *term* is either a *pair* of terms, a *constant*, or a *variable*. A variable exists in one of four states: It can either be unbound, or *ultimately* bound to an unbound variable, to a constant, or to a pair. When we say "ultimately bound", we mean bound through a chain of variable bindings.

A unifier program normally takes two input parameters x_0 and y_0, which are two terms to unify. If unification is successful, the program updates the variable bindings and returns the Boolean value *true*. If unification fails, the program returns *false*.

We assume that a possible state representation for such a program would be a pair of state variables x and y, initially set to x_0 and y_0.

We have chosen the following primitive operations as a relevant selection:

Terminals:
FAIL: Returns from the program with the value *false*. This operation may only be applied to a state if x is a pair and y a constant, or vice versa.
EQUAL: Returns *true* if $x = y$, else *false*. x and y must both be constants.
BINDXY: Binds x to y, unless x is identical to y. x must be an unbound variable. y must not be a bound variable. Returns *true*. Note that the binding process is internal to this primitive. How internal things like trailing etc. are handled inside the operations does not affect the specification.
BINDYX: Similar.
RECURSE: Calls the unifier program recursively on the right and left subtrees of x and y. Returns *true* if both of the recursive calls do. x and y must both be pairs.

An additional precondition (the correctness criterion) for all terminals is that the state variables x and y must have either their initial values, or their initial values *dereferenced*. I.e. if the initial value x_0 is a variable, x_0 dereferenced is what this variable ultimately is bound to.

Transformations:
DEREFX: Dereference the variable x. x must be a bound variable.
DEREFY: Similar.

Tests:
UNBOUNDX: Chooses the left successor if x is unbound, else the right one. x must be a variable.
UNBOUNDY: Similar.
CONSTX: Chooses the left successor if x is a constant, else the right one.
CONSTY: Similar.
NOTVARX: Chooses the left successor if x isn't a variable, else the right one. x must not be a constant.
NOTVARY: Similar.

5. Generating all unifiers

We shall first simplify the specification of the primitive operations. Then we shall translate the specification to a Prolog program that generates all unifier programs.

The correctness criterion requires the state variables not to be changed from their initial values, except that they may be dereferenced. The state variables can only be changed by transformation operations. Since the only transformations available dereference the state variables, this criterion is always satisfied, and need not be checked in the Prolog program.

Since we use two state variables x and y, the program's state is a tuple (x, y). We shall divide the set of possible states in subclasses to simplify the specification of pre- and postconditions. Let us recall that x and y represent memory cells in the program. These cells contain some data structures. They could belong to, for instance, the set of constants. We denote this set S_c. The structures could also belong to the set of unbound variables, which we denote by S_*; or the set of all pairs, S_p; the set of all variables ultimately bound to constants, S_C; the set of all variables ultimately bound to pairs S_P; or, finally, the set of all variables ultimately bound to unbound variables, S_V. For convenience, let S_b be the set of all variables that are bound to something, even if it is an unbound variable, $S_b = S_C \cup S_P \cup S_V$. The set $S = S_p \cup S_c \cup S_* \cup S_b$ then contains all possible data structures. The set of possible program states is $S \times S$:

$$\forall si\ (correctstateset(si) \leftrightarrow si \subseteq S \times S)$$

There is an important observation that simplifies the Prolog program considerably: All preconditions care only about which *types* of data structures (constants, variables bound to pairs, etc.) the state variables hold. Therefore, any details in the postconditions beyond those specifying the classification of the output state, are unnecessary.

The simplified logical specification of operations will then be:

$$\forall si\ (terminal(si, FAIL) \leftrightarrow si \neq \emptyset \ \wedge\ si \subseteq (S_c \times S_p) \cup (S_p \times S_c))$$

$$\forall si\ (terminal(si, EQUAL) \leftrightarrow si \neq \emptyset \ \wedge\ si \subseteq S_c \times S_c)$$

$$\forall si\ (terminal(si, BINDXY) \leftrightarrow si \neq \emptyset \ \wedge\ si \subseteq S_* \times (S \setminus S_b))$$

$$\forall si\ (terminal(si, BINDYX) \leftrightarrow si \neq \emptyset \ \wedge\ si \subseteq (S \setminus S_b) \times S_*)$$

$$\forall si\ (terminal(si, RECURSE) \leftrightarrow si \neq \emptyset \ \wedge\ si \subseteq S_p \times S_p)$$

$$\forall si, so\ (transformation(si, DEREFX, so) \leftrightarrow so = (S_b \times S) \cap si \ \wedge\ si \neq \emptyset)$$

$$\forall si, so\ (transformation(si, DEREFY, so) \leftrightarrow$$

$$so = (S \times S_b) \bigcap si \ \wedge \ si \neq \emptyset)$$

$$\forall si, so_1, so_2 \ (test(si, UNBOUNDX, so_1, so_2) \leftrightarrow$$
$$si \subseteq (S_b \bigcup S_v) \times S \ \wedge$$
$$so_1 = (S_v \times S) \bigcap si \ \wedge \ so_1 \neq \emptyset \ \wedge$$
$$so_2 = (S_b \times S) \bigcap si \ \wedge \ so_2 \neq \emptyset)$$

$$\forall si, so_1, so_2 \ (test(si, UNBOUNDY, so_1, so_2) \leftrightarrow$$
$$si \subseteq S \times (S_b \bigcup S_v) \ \wedge$$
$$so_1 = (S \times S_v) \bigcap si \ \wedge \ so_1 \neq \emptyset \ \wedge$$
$$so_2 = (S \times S_b) \bigcap si \ \wedge \ so_2 \neq \emptyset)$$

$$\forall si, so_1, so_2 \ (test(si, CONSTX, so_1, so_2) \leftrightarrow$$
$$so_1 = (S_c \times S) \bigcap si \ \wedge \ so_1 \neq \emptyset \ \wedge$$
$$so_2 = ((S \setminus S_c) \times S) \bigcap si \ \wedge \ so_2 \neq \emptyset)$$

$$\forall si, so_1, so_2 \ (test(si, CONSTY, so_1, so_2) \leftrightarrow$$
$$so_1 = (S \times S_c) \bigcap si \ \wedge \ so_1 \neq \emptyset \ \wedge$$
$$so_2 = (S \times \ (S \setminus S_c)) \bigcap si \ \wedge \ so_2 \neq \emptyset)$$

$$\forall si, so_1, so_2 \ (test(si, NOTVARX, so_1, so_2) \leftrightarrow$$
$$si \subseteq (S \setminus S_c) \times S \ \wedge$$
$$so_1 = ((S_v \bigcup S_b) \times S) \bigcap si \ \wedge \ so_1 \neq \emptyset \ \wedge$$
$$so_2 = (S_p \times S) \bigcap si \ \wedge \ so_2 \neq \emptyset)$$

$$\forall si, so_1, so_2 \ (test(si, NOTVARY, so_1, so_2) \leftrightarrow$$
$$si \subseteq S \times \ (S \setminus S_c) \ \wedge$$
$$so_1 = (S \times \ (S_v \bigcup S_b)) \bigcap si \ \wedge \ so_1 \neq \emptyset \ \wedge$$
$$so_2 = (S \times \ S_p) \bigcap si \ \wedge \ so_2 \neq \emptyset)$$

When we translate the specification to Prolog, the following statements will be taken care of by the negation–as–failure rule:

$$\forall si, op \ (\neg \ terminal(si, op) \leftarrow$$
$$op \notin \{FAIL, EQUAL, BINDXY, BINDYX, RECURSE\})$$

$$\forall si, so, op \ (\neg \ transformation(si, op, so) \leftarrow$$
$$op \notin \{DEREFX, DEREFY\})$$

$$\forall si, op, so_1, so_2 \ (\neg \ test(si, op, so_1, so_2) \leftarrow$$
$$op \notin \{UNBOUNDX, UNBOUNDY, CONSTX, CONSTY, NOTVARX, NOTVARY\})$$

Every test and transformation operation deals with one state variable only. This suggests a compact Prolog encoding of the sets of states: Let the Prolog lists of constants $\{b\}$, $\{v\}$, $\{c\}$, and $\{p\}$ denote the sets S_b, S_v, S_c, and S_p. Unions and intersections correspond straightforwardly to lists. E.g., $S_c \bigcup S_p$ is encoded as $\{c,p\}$. A Cartesian product of sets is encoded as a tuple of lists: $S_c \times (S_c \bigcup S_p)$ corresponds to $(\{c\},\{c,p\})$.

The *program*-predicate can now easily be translated to Prolog (upper case symbols are Prolog variables):

```
program(SI, OP) ← terminal(SI, OP).
program(SI, sequence(OP, P1)) ←
      transformation(SI, OP, SO) ∧
      program(SO, P1).
program(SI, if(OP, P1, P2)) ←
      test(SI, OP, SO1, SO2) ∧
      program(SO1, P1) ∧
      program(SO2, P2).
```

The Prolog specification for the *FAIL* operation looks like:

```
terminal((({p}, {c}), fail).
terminal((({c}, {p}), fail).
```

The first of these clauses says that the *FAIL* operation accepts an input state $(x, y) \in S_p \times S_c$. Specifications for the other operations are similar:

```
terminal((({c}, {c}), equal).
terminal((({v}, Y), bindxy) ←
      intersection(Y, {b}, {}).
terminal((X, {v}), bindyx) ←
      intersection(X, {b}, {}).
terminal((({p}, {p}), recurse).

transformation((({b}, Y), derefx, (({v, c, p}, Y)).
transformation((X, {b}), derefy, (X, {v, c, p})).

test((({b, v}, Y), unboundx, (({v}, Y), (({b}, Y)).
test((X, {b, v}), unboundy, (X, {v}), (X, {b})).
test((X, Y), constx, (X1, Y), (X2, Y)) ←
      intersection(X, {c}, X1) ∧ X1 ≠ ∅ ∧
      intersection(X, {b, v, p}, X2) ∧ X2 ≠ ∅.
```

$test((X,Y),consty,(X,Y1),(X,Y2)) \leftarrow$
$\quad intersection(Y,\{c\},Y1) \wedge Y1 \neq \emptyset \wedge$
$\quad intersection(Y,\{b,v,p\},Y2) \wedge Y2 \neq \emptyset.$
$test((X,Y),notvarx,(X1,Y),(X2,Y)) \leftarrow$
$\quad intersection(X,\{c\},\{\}) \wedge$
$\quad intersection(X,\{b,v\},X1) \wedge X1 \neq \emptyset \wedge$
$\quad intersection(X,\{p\},X2) \wedge X2 \neq \emptyset.$
$test((X,Y),notvary,(X,Y1),(X,Y2)) \leftarrow$
$\quad intersection(Y,\{c\},\{\}) \wedge$
$\quad intersection(Y,\{b,v\},Y1) \wedge Y1 \neq \emptyset \wedge$
$\quad intersection(Y,\{p\},Y2) \wedge Y2 \neq \emptyset.$

Consider the sample Prolog call

$$program((\{c,p\},\{c,p\}),P)$$

The Prolog program will instantiate P to different unifier programs on the premise that x_0 and y_0 are constants or pairs. Two possible programs P are generated:

$$if(constx,if(consty,equal,fail),$$
$$\quad if(consty,fail,recurse))$$

and

$$if(consty,if(constx,equal,fail),$$
$$\quad if(constx,fail,recurse))$$

The call

$$program((\{b,v,c,p\},\{b,v,c,p\}),P)$$

will generate all different unifier programs. If we assume that all primitive operations require only a finite amount of time, the generated programs will be correct.

6. Finding an optimal unifier

The Prolog program from the previous section can now be modified so that a statistically expected cost – here, the execution time – is estimated along the generation of the unifiers. To do this, we introduce *cost* parameters and *frequency tables* in the parameter lists of the predicates.

A frequency table is a list of 36 numbers. They are the frequencies of the $6 \times 6 = 36$ different combinations of types of state variables: x belongs to one of S_V, S_C, S_P, S_*, S_ϵ, S_γ, and similarly for y. At every node, the current frequency table is summed up, and multiplied with the execution time of this node's operation, in order to compute the expected cost. The primitive operations affect the table: For instance, a *DEREFX* operation changes all $(S_C \times S_\gamma)$-states to $(S_* \times S_\gamma)$-states. The frequency of the former kind of state will be zero after the operation has been performed. The frequency of the latter kind will increase with the same amount as the frequency of the former decreased. Another example is a test operation, which splits the frequency table into two new tables: one for the left successor and one for the right successor.

The *COST* parameter holds the accumulated expected cost. The predicate *sum* adds up the total number of different cases in the frequency table.

The *FREQ* parameter holds the "input" frequency table, and the *FREQ*1 and *FREQ*2 parameters hold the "output" tables.

$program(SI,OP,FREQ,COST) \leftarrow$
$\quad terminal(SI,OP,FREQ) \wedge$
$\quad cost(OP,C) \wedge$
$\quad sum(FREQ,N) \wedge$
$\quad COST\ is\ C * N.$

$program(SI,sequence(OP,P1),$
$\quad FREQ,COST) \leftarrow$
$\quad transformation(SI,OP,SO,$
$\quad\quad FREQ,FREQ1) \wedge$
$\quad program(SO,P1,FREQ1,COST1) \wedge$
$\quad cost(OP,C) \wedge$
$\quad sum(FREQ,N) \wedge$
$\quad COST\ is\ C * N + COST1.$

```
program(SI, if(OP, P1, P2),
        FREQ, COST) ←
    test(SI, OP, SO1, SO2,
        FREQ, FREQ1, FREQ2) ∧
    program(SO1, P1, FREQ1, COST1) ∧
    program(SO2, P2, FREQ2, COST2) ∧
    cost(OP, C) ∧
    sum(FREQ, N) ∧
    COST is C * N + COST1 + COST2.
```

There should be one clause *cost*(OP, C) for every operation OP. C is the required execution time for OP. The call *sum*$(FREQ, N)$ binds N to the sum of all numbers in the table *FREQ*.

We assume that an operation requires constant time, regardless of the state it operates upon. (This has shown to be a reasonable approximation for our test implementation of the primitive operations.) There is one exception to this: The *RECURSE* operation. However, if N is the number of calls to the unifier, and T is the expected execution time of executing one step of the unifier *without recursion*, the expected cost of the complete unification will be NT.

Suppose that we have generated two unifier programs P_1 and P_2 for some set of input states. Suppose also that the execution of P_1 ends with a *BIND* (i.e. *BINDXY* or *BINDYX*) operation, given some particular input state. Then the execution of P_2 must also end with a *BIND* operation for the same input. Likewise, if P_1 ends with *FAIL*, P_2 also ends with *FAIL*. The same thing holds for the *EQUAL* and *RECURSE* operations as well. That is to say, if P_1 spends an expected time T_0 executing terminal operations, then P_2 will also spend the expected time T_0 at terminal operations.

For the program P_1, our unifier generator will compute $T_1 - T_0$, where T_1 is the expected time of one step of the unifier without recursion. If $T_2 - T_0$ is the value computed for P_2, the relation between these two values provides us with an *upper bound* on the speed difference. Without restriction, assume that P_1 is faster than P_2 $(T_1 > T_2)$:

$$\frac{T_1 - T_0}{T_2 - T_0} = \frac{T_1}{T_2} + T_0 \frac{(T_1/T_2 - 1)}{T_2 - T_0} > \frac{NT_1}{NT_2} > 1.$$

Here are some examples of what the unifier generator looks like. We do not list all the clauses for all operations since the rest of the program does not contain anything essential beyond what is given here.

The Prolog specification for the *FAIL* operation looks like this:

```
terminal(({p}, {c}), fail, FREQ).
terminal(({c}, {p}), fail, FREQ).
```

The *FREQ* variable is just a dummy in *terminal* clauses. In *transformation* and *test* clauses, however, is passes the frequency table, which is in the following format:

```
{FVV, FVC, FVP, FVv, FVc, FVp,
 FCV, FCC, FCP, FCv, FCc, FCp,
 FPV, FPC, FPP, FPv, FPc, FPp,
 FvV, FvC, FvP, Fvv, Fvc, Fvp,
 FcV, FcC, FcP, Fcv, Fcc, Fcp,
 FpV, FpC, FpP, Fpv, Fpc, Fpp}
```

The element Fij is the frequency of the states $(x, y) \in S_i \times S_j$. Example: *FPc* is the frequency of the state where x holds a variable that is bound to a pair, and y holds a constant.

The specification for *DEREFX* is

```
transformation(({b}, Y), derefx, ({v, c, p}, Y),
        {FVV, FVC, FVP, FVv, FVc, FVp,
         FCV, FCC, FCP, FCv, FCc, FCp,
         FPV, FPC, FPP, FPv, FPc, FPp,
          0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,
```

$$
\begin{array}{llllll}
0, & 0, & 0, & 0, & 0, & 0\}, \\
\{0, & 0, & 0, & 0, & 0, & 0, \\
0, & 0, & 0, & 0, & 0, & 0, \\
0, & 0, & 0, & 0, & 0, & 0, \\
FVV, & FVC, & FVP, & FVv, & FVc, & FVp, \\
FCV, & FCC, & FCP, & FCv, & FCc, & FCp, \\
FPV, & FPC, & FPP, & FPv, & FPc, & FPp\}).
\end{array}
$$

For *CONSTY*, it looks like this:

$$
\begin{array}{llllll}
test((X,Y), consty, (X,Y1), (X,Y2), & & & & & \\
\{FVV, & FVC, & FVP, & FVv, & FVc, & FVp, \\
FCV, & FCC, & FCP, & FCv, & FCc, & FCp, \\
FPV, & FPC, & FPP, & FPv, & FPc, & FPp, \\
FvV, & FvC, & FvP, & Fvv, & Fvc, & Fvp, \\
FcV, & FcC, & FcP, & Fcv, & Fcc, & Fcp, \\
FpV, & FpC, & FpP, & Fpv, & Fpc, & Fpp\}, \\
\{0, & 0, & 0, & 0, & FVc, & 0, \\
0, & 0, & 0, & 0, & FCc, & 0, \\
0, & 0, & 0, & 0, & FPc, & 0, \\
0, & 0, & 0, & 0, & Fvc, & 0, \\
0, & 0, & 0, & 0, & Fcc, & 0, \\
0, & 0, & 0, & 0, & Fpc, & 0\}, \\
\{FVV, & FVC, & FVP, & FVv, & 0, & FVp, \\
FCV, & FCC, & FCP, & FCv, & 0, & FCp, \\
FPV, & FPC, & FPP, & FPv, & 0, & FPp, \\
FvV, & FvC, & FvP, & Fvv, & 0, & Fvp, \\
FcV, & FcC, & FcP, & Fcv, & 0, & Fcp, \\
FpV, & FpC, & FpP, & Fpv, & 0, & Fpp\}) \leftarrow & & & & \\
intersection(Y, \{c\}, Y1) \wedge Y1 \neq \emptyset \wedge & & & & & \\
intersection(Y, \{b, v, p\}, Y2) \wedge Y2 \neq \emptyset. & & & & &
\end{array}
$$

7. Results

We have made a simple test implementation of the primitive operations to try out the optimization method.

The *naive-reverse* benchmark is a common way to measure the efficiency of Prolog systems (Warren 77). It was used to compute frequencies of different types of parameters to unify:

		y_0					
		V	*C*	*P*	*v*	*c*	*p*
	V	0	0	0	0	0	0
	C	0	0	0	0	0	0
x_0	*P*	1	0	0	29	0	0
	v	0	435	841	30	60	59
	c	0	31	464	1	496	1
	p	29	31	464	406	0	1954

The execution times of the transformation and the test primitives were approximately

operation	*cost* (μs)
DEREFX, *DEREFY*	75
CONSTX, *CONSTY*	4
UNBOUNDX, *UNBOUNDY*	40
NOTVARX, *NOTVARY*	5

When the Prolog call

$$
program((\{b, v, c, p\}, \{b, v, c, p\}), P, FREQ, COST)
$$

was executed with *FREQ* instantiated according to the frequency table above, the difference between the maximum *COST* and the minimum *COST*, was about 2%. If *FREQ* was instantiated to a table of uniform frequencies the difference was 4.5%. If the costs of the transformations and tests were all set to *one*, and *FREQ* instantiated according to the frequency table, the difference was around 8%. With uniform frequencies the difference was 2%.

No significant difference could be measured between actual implementations of a worst case and a best case unifier program when the *naive-reverse* benchmark was run.

8. Discussion

The differences in speed between the best and the worst unifiers will be smaller than the values computed in the previous section, since the execution time of the terminal operations is excluded. The speed differences of Prolog interpreters using those unifiers will be even less.

Even in such a case as *naive-reverse*, with a very non-symmetric distribution of types of arguments, it seems to matter very little what the order of the unifier's primitive operations are. However, it should be remembered that the situation might be different for a unifier with other primitive operations. Maybe the most severe restriction of our set of primitives is that no "multi-way conditional" exists. Such an operation can be used to dispatch very efficiently on data type tags, and will increase the speed of the unifier substantially.

One of the anonymous referees of this paper suggested that our method could be used in a compiler for finding fast unification code. A typical situation for a Prolog compiler which tries to opencode unification is that something is known about the types of the terms to be unified. The compiler's task is to use that knowledge to find the fastest and smallest sequences of instructions which perform the unification.

The method described in this paper seems to be useful for optimizing other kinds of small programs, too. However, a hard problem is that the set of generated programs easily grows far too large. It becomes impossible to find the optimal programs by pure depth-first search. An approach that might prove to be valuable in the future would be to use a best-first search based on the accumulated cost.

9. Acknowledgments

I would like to thank my colleagues at UPMAIL for the help to proofread this article. I am also grateful to the anonymous referees for useful suggestions, to Annika Ström for checking the English of part of the paper, and to Peter Löthberg for improving the printing quality.

10. References

Eriksson, L.-H.: *Synthesis of a Unification Algorithm in a Logical Programming Calculus*. Technical Report No. 22. UPMAIL, Computing Science Department, Uppsala University, Sweden, August 1983.

Frisch, A., Allen, J., Giuliano, M.: *An Overview of the Horne Logic Programming System*. Logic Programming Newsletter No. 5, p 2–3. Winter 1983/1984.

Nilsson, M.: *FOOLOG – A Small and Efficient Prolog Interpreter*. Technical Report No. 20. UPMAIL, Computing Science Department, Uppsala University, Sweden, June 1983.

Warren, D. H. D.: *Implementing Prolog - compiling predicate logic programs*. D.A.I. Research Report No. 39, 40. Department of Artificial Intelligence, University of Edinburgh, May 1977.

DRAWING TREES AND THEIR EQUATIONS IN PROLOG

Jean Francois Pique
Groupe Representation et
Traitement des Connaissances
C.N.R.S.
31 chemin Joseph Aiguier
13402 Marseille CEDEX 9,France

ABSTRACT

This paper describes in detail how to compute efficiently a drawing of Prolog trees with the smallest number of nodes. This is done using a system of equations as in Colmerauer (1982). We give examples with finite and infinite trees in different domains.

1.0 INTRODUCTION

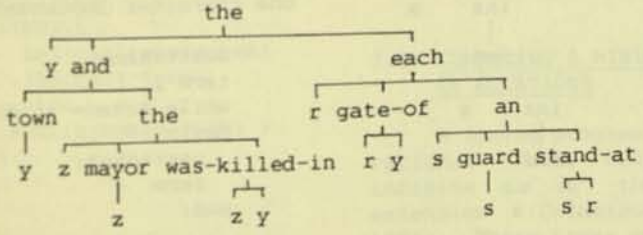
When handling complex trees, the usual functional notation is really a maze, and is a major cause of mistakes. When creating natural language front ends in Prolog, I have regretted the lack of a more visual representation of trees which is the main difficulty in grammar debugging. This was the original motivation for building the tools described here. Colmerauer's modification (Colmerauer 1982) of the theoretical model of Prolog, while adding the complexity of infinite trees, introduces powerful ideas for tree representation optimisation: mainly, to define a tree with a system of equations with the smallest number of symbols. This point is described in detail in the second chapter.

Of all the possible representations (functional, indentation, ...) of terms, the graphical representation of the arborescence is by far the clearest and the most pleasant, although the most difficult to manage. A convenient algorithm to draw finite trees in a compact manner is described in the third chapter.

In what follows are given examples in three different domains. The first one (Fig. 1) demonstrate the semantic tree (Pique 1982) obtained in the analysis of the sentence:

"A guard is standing at each gate of the town where the mayor was killed"

```
>draw-tree(the(y,and(town(y),the(z,mayor(z),was-killed-in(z,y))
   ),each(r,gate-of(r,y),an(s,guard(s),stand-at(s,r)))));
```

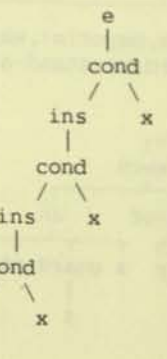


NOTE ON PROLOG II SYNTAX

Before continuing, some remark on the Prolog II syntax is worth noting: constant symbols begin with two letters, while variable symbols begin with only one letter eventually followed by digits and single quotes. An hyphen may appear inside a symbol. A semicolon ends a rule. Lists are written with infix dot notation. Terms may be written as functions e.g. ff(a1,...,an) or as tuples e.g. <ff,a1,...,an>. These two notations are equivalent, however the first one is only allowed when the first element of the tuple is an identifier. With the tuple representation one can do very fast and easy term composition and decompostion.

The next example shows the output of a compiler for a structured language like Pascal. This compiler compiles loops into infinite trees of code instructions. Each structured instruction has only one entry point "e" and one exit point "x". The same is true for the generated code, except for the conditional branch which has two exit points: the left one is the true condition exit point, the right one is the false condition exit point. As an example, for the structured instruction "while", we get:

```
while cond do
begin
  ins
end
```



Compiling is nothing but the connection of code instruction trees:

```
compile(WHILE(c,i),e,x) ->
  compile-test(c,e,x1,x2)
  compile(i,e',x')
  equal(x1,e')
  equal(x',e)
  equal(x2,x);
compile(REPEATUNTIL(i,c),e,x) ->
  compile(i,e,x')
  compile-test(c,e',x1,x2)
  equal(x',e')
  equal(x2,e)
  equal(x1,x);
compile(IF(c,i1,i2),e,x) ->
  compile-test(c,e,x1,x2)
  compile(i1,e1,x)
  compile(i2,e2,x)
  equal(x1,e1)
  equal(x2,e2);
compile(INS(i),e,x) ->
  equal(e,<i,x>);
compile(nil,e,x) ->
  equal(e,x);
compile(i.l,e,x) ->
  compile(i,e,x')
  compile(l,e',x)
  equal(x',e');

compile-test(NOT(c),e,x1,x2) ->
  compile-test(c,e,x2,x1);
compile-test(c,e,x1,x2) ->
  atomic(c)
  equal(e,<c,x1,x2>);

equal(y,y) ->;
```

Consider now a fragment of a classical program to parse an expression "term {+ term}" with one character lookahead:

```
nexttoken;
term ;
while token='+' do
begin
  nexttoken;
  term
end;
```

It is interesting to compile and then draw the solution:

```
>compile
  (  INS(call-nexttoken)
   .INS(call-term)
   .WHILE(is-token-PLUS
         , INS(call-nexttoken)
          .INS(call-term)
          .nil )
   .nil
  , e
  , x )
draw-equ( e );
     e = call-nexttoken
                |
          call-term
                |
       token-is-PLUS
              ┌─┴─┐
              e   x
```

As one can see, the solution is very simple, exhibiting a minimum code sequence. We can also compare with the alternate "repeat" solution in structured programming:

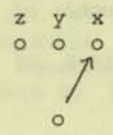
```
nexttoken;
term ;
if token='+' then
  repeat
    nexttoken;
    term
  until not (token='+');
```

which, when compiled, leads to the same infinite tree:

```
>compile
  ( INS(call-nexttoken)
   .INS(call-term)
   .IF(is-token-PLUS
      , REPEATUNTIL
           (  INS(call-nexttoken)
            .INS(call-term)
            .nil
           , NOT(is-token-PLUS) )
      , nil )
   .nil
  , e
  , x )
```

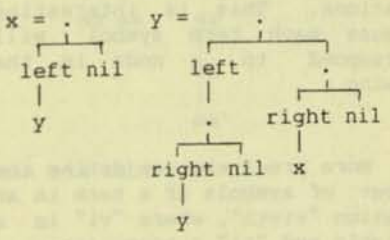
```
draw-equ( e );
     e = call-nexttoken
                |
          call-term
                |
       token-is-PLUS
              ┌─┴─┐
              e   x
```

The third example defines the transition diagram of a three state switch. Each state is described by a list of pairs (transition, new state), nil meaning no transition. The initial state is "x":




```
switch(x) ->
  equal(x,<left,y>.nil)
  equal(y,<left,z>.<right,x>.nil)
  equal(z,<right,y>.nil);
```

```
>switch(x)  draw-equ(x);
```



2.0 COMPUTING A MINIMAL SYSTEM OF EQUATIONS

A Prolog program manipulates rational trees, finite or even infinite as in the Marseille extension (Colmerauer and al 1981). These trees are defined in

the new theoretical model by a system of equations, hence comes the idea to compute from the tree a pleasant representation of a system defining it. Colmerauer has given in his paper (Colmerauer 82) a program to do this. I describe in detail here an efficient program which computes a system built from the functional symbols of the tree and a set of variables. A good looking system is one fulfilling the following conditions:

- A minimum number of equations
- No duplication of non atomic terms.

The second condition enhances the ability to identify identical complex terms. It can be proved that, among the equivalent systems, it is minimal in the number of symbols of FuV occuring in the right member of the equations. This is interesting because each term symbol will correspond to a node in the drawing.

More precisely, we define the number of symbols of a term in an equation "vi=ti", where "vi" is a variable and "ti" a term, as:

1. If "ti" is a constant or a variable : one.

2. If "ti" is of the form "fn(t1,...,tn)", where "fn" is a member of Fn, and "t1,...,tn" are terms : one plus the sum of the number of symbols of the terms "t1,...,tn".

The number of symbols of the terms of a system is then the sum of the numbers from each equation. The system is however not minimal if all symbols of the equation are counted as can be seen from the following two equivalent systems :

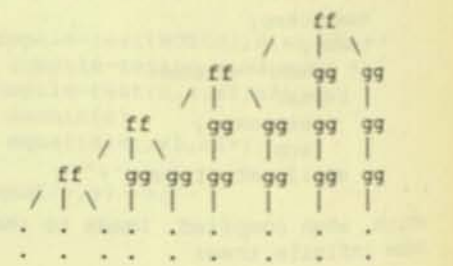
{x=f(y,y), y=g(a)}

{x=f(g(a),g(a))}

As an example, consider the tree defined by the following Prolog program:

```
tree( x ) ->
  equal( x, ff(u,y,z) )
  equal( y, gg(a) )
  equal( z, gg(b) )
  equal( b, gg(a) )
  equal( a, gg(b) )
  equal( u, ff(ff(x,z,a),a,b) );

equal( x, x ) -> ;
```

which says that the following tree is a member of the set of assertions:



A system of equations defining this tree and satisfying our criterions (it is also minimal with the second definition) is for example:

{x=ff(x,y,y), y=gg(y)}

Since terms in the equation are finite, they can be drawn as arborescences, enhancing further readability. As a consequence of minimality, even finite trees may gain a plus from this representa-

tion. First consider completely specified trees (i.e. with no variables). We will assume that the first equation in the system defines the root of the tree.

The algorithm consists in building a basic system with one equation per subtree, and then reducing it. Remember that Prolog programs can only define rational trees (i.e. trees with a finite set of subtrees), so this basic system always has a finite number of equations. To build this basic system, first pair each different subtree with a different variable symbol of the system. Then the system is easily constructed as follows: For each different subtree add to the system the equation "v=f(v1,...,vn)" where "v" is the variable symbol paired with the subtree, "f" is the functional symbol of the subtree, and "v1,...,vn" are the variable symbols paired with the sons of the subtree.

For example, the preceding tree has two subtrees: itself, and the tree "gg(gg(...))". If we pair them with the symbol variables "x" and "y" respectively, we obtain the system already seen:

```
{x=ff(x,y,y), y=gg(y)}
```

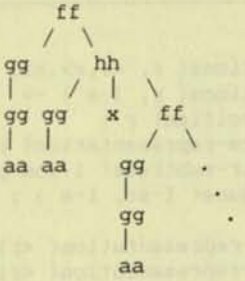
In case of an uncompletely specified tree (i.e. a tree containing variables) we must define the exact meaning of "different subtrees". We say that two uncompletely specified subtrees are different if there exists a tree assignment such that they are different, which is what formal inequality involves. We may thus consider tree variables as constants different from those of the tree, and formally represented by variable symbols not occurring in the system. Different variables are different constants paired with different variable symbols.

Now, to reduce the system S, we consider every equation "vi=ti" of the system except the first one. There are two reduction conditions:

1. If "ti" has only one symbol, remove the equation and replace each occurrence of "vi" in S by an occurrence of "ti".

2. If "ti" has more than one symbol, "vi" has no occurrence in "ti", and there is only one occurrence of "vi" in the system when the equation is removed, remove the equation, and replace this occurrence by an occurrence of "ti".

For example the tree



has six subtrees:

```
x
aa
gg(aa)
gg(gg(aa))
hh(gg(aa),x,ff(...))
ff(gg(gg(aa)),hh(...))
```

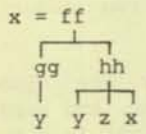
We get the basic system:

```
{ x0 = ff(x1,x4),
  x1 = gg(x2),
  x2 = gg(x3),
  x3 = aa,
  x4 = hh(x2,x5,x0) }
```

after reduction:

```
{ x0 = ff(gg(x2),hh(x2,x5,x1)),
  x2 = gg(aa) }
```

using our program to draw the system, we get:



In our program we use Prolog variables as system variable symbols. This allow very fast reduction because replacement of occurrences can be done by unification. While computing each subtree, we also pair a new variable symbol, create the corresponding equation and carry out the first type of reduction. Subtrees with multiple occurrences are also flagged, so that the final reduction stage is straight forward. Each equation is represented by a 2-uple "<v,t>" where "v" is a variable and "t" a term. The system is represented as a list of equations, the first one being the last in the list. Rational trees are denoted "r", "st" is a triplet (subtree, equation, number of subtree occurences as immediate son of a subtree), "pair-subtrees" take a pair "(v,r)" and add the new subtrees in it to the list of subtrees. Lists variables begin with "l-", e.g. "l-st" is a variable standing for a list of subtree triplets.

```
equations( r, <v,r>.nil ) -> constant( r ) ;
equations( r, l-e ) ->
  specified( r )
  term-representation( r, t, l-son-pairs )
  pair-subtrees( l-son-pairs, st(r,<v,t>,1).nil, l-st )
  reduce( l-st, l-e ) ;

term-representation( <r1>, <v1>, pair(v1,r1).nil ) -> ;
term-representation( <r1,r2>, <v1,v2>
                   , pair(v1,r1).pair(v2,r2).nil ) -> ;
....


pair-subtrees( nil, l-st, l-st ) -> ;
pair-subtrees( p.l-p, l-st, l-st' ) ->
  subtrees( p, l-st, l-st1 )
  pair-subtrees( l-p, l-st1,l-st' ) ;

subtrees( pair(v,r), l-st, l-st ) ->
  constant( r )
  substitute( v, r ) ;
subtrees( pair(v,r), l-st, l-st' ) ->
  specified( r )
  in-list( r, l-st, c, l-st1 )
  add-subtree( c, pair(v,r), l-st1, l-st' ) ;
```

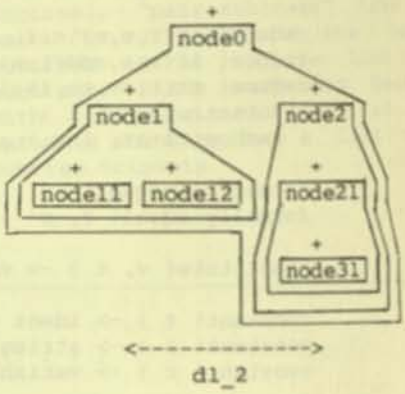
```
in-list( r, nil,  not-in, nil ) -> ;
in-list( r, st(r',<v,t>,n).l-st, named(v)
       , st(r',<v,t>,add(l,n)).l-st  )   ->
  formally-equal( r, r' ) ;
in-list( r, s.l-st, c, s.l-st' ) ->
  eg( s, st(r',e,n) )
  formally-inequal( r, r' )
  in-list( r, l-st, c, l-st' ) ;

add-subtree( not-in, pair(v,r), l-st, l-st' ) ->
  term-representation( r, t, l-p )
  pair-subtrees( l-p, st(r,<v,t>,l).l-st, l-st' ) ;
add-subtree( named(vl), pair(v,r), l-st, l-st ) ->
  substitute( v, vl ) ;

reduce( st(r,e,n).nil, e.nil ) -> ;
reduce( st(r,e,add(l,n)).l-st, e.l-e ) -> reduce( l-st, l-e ) ;
reduce( st(r,<v,t>,l).l-st, l-e ) ->
  substitute( v, t )
  reduce( l-st, l-e ) ;

formally-inequal( r, r' ) -> dif( r, r' ) ;
formally-equal( r, r' ) -> not( dif(r,r') ) ;

substitute( v, t ) -> variable( v )  eg( v, t ) ;

constant( r ) -> ident ( r ) ;
constant( r ) -> string( r ) ;
constant( r ) -> variable( r ) ;

specified( r ) -> not( variable(r) ) ;
```

3.0 DRAWING OF A FINITE TREE

We must first examine some conditions on the way we want things to work. First, we do not want to set any bounds on the depth of the drawn trees. Second, we need as much independance as possible from the kind of hardware on hand (paper teletype, video,..), although we want to be allowed to benefit from graphic features when these exist. Third and last, we want to represent the tree with as much compactness as is compatible with good readability. The first two conditions imply to output the tree line by line, starting from the root. To fulfil the third one, the bidimensional optimisation of the nodes' placement was rejected, because it is too complex and costly. We prefer to place each node with the same depth level on the same line, as close as possible not to overlap their arborescences. A subsidiary advantage is that the ramification symbols are kept apart on the same line, allowing the use of semi-graphical possibilities when present (the program which made the drawings in this paper has a data base defined for the DEC VT100 terminal and an Epson FX80 hardcopy).

A survey of the problem sets up an interesting dilemma : how do we do determine the position of, say, the root? This position will depend on the other node positions, but they too will each depend on others (every node ramification may arbitrarily extend to the left or to the right), giving an appearance of circularity in which, as with the egg and the hen, one does not know where to start.

The solution is to keep separate the problem of the absolute position of the nodes, from that of their relative horizontal distance. Then, the distance between two nodes will be the minimum one such that there is no overlap of their branches. Knowing the relative distance between all the sons of a node, we can compute their relative distance from this node. We thus have a bottom up process which, starting from the leaves, allows for each node the computation of its deviation from the father.




```
g11_1 := d11_12 / 2
g1_0  := d1_2 / 2
```

The key point is therefore to determine the non overlapping condition. To do this, we compute for each node, a list of the maximum width of its subtrees at each depth level. These lists can be computed during the same bottom up process: knowing the relative widths for each son of a node, we get the node width by merging the lists from the sons, with the necessary shift. Once we know all widths from the root, we can compute its position with regard to the margins, and then that of every node. In the program, each son deviation is relative to the position of the first son. The computation of their absolute position is coroutined on the evaluation of the first son

position which in turn is coroutined on the evaluation of the position of the father. So, all absolute position computations are delayed until the position of the root is determined from the tree width in the "margin" rule. The tree is then drawn top down breadthfirst. To do that, each time a slice of nodes is printed, the list of sons' descriptions are concatenated. This can be fast when using difference lists (for information on d-lists see: Clark K.L. an Tarnlund S.A. 1977) Here is an outline of the program. "g" (left) and "r" (right) denote the extreme deviations of an arborescence at a given level, "d" is the deviation of a node from his leftmost brother, "x" the absolute position of a node, "t" a tree and "t'" the description of his drawing, "o" is the printing sequence of a node symbol and "s" his size. Lists variables begin with an "1", so "l-t" is a variable representing a list of trees. Difference list variables are denoted "d-l", and "<l,q>" denotes a d-list starting at "l" ending at "q". "l-w" is the list of extreme deviations (i.e. pair "(g.r)") of a subtree until maximum depth level.

```
draw-finite-tree( t ) ->
  node-positions( t, x,  t', l-w )
  margin( x, l-w )
  by-slice( t'.nil ) ;

node-positions( t, x, node(x,<o,g>), (g.r).nil ) ->
  atom( t, o, s )
  center-node( s, g, r ) ;
node-positions( t, x, node(x,<o,g,n,d-l>), (g.r).l-w' ) ->
  specified( t )
  tree-split( t, o, s, n, l-sons )
  center-node( s, g, r )
  son-positions( l-sons, xl, nil,  d-l, l-w, d-max )
  value( dl, sub(0,div(d-max,2)) )
  shift( l-w, dl, l-w' )
  freeze-on( x, value( xl, add(dl,x) ) ) ;

center-node( s, g, r ) ->
  value( r, div(s,2) )
  value( g, sub(add(r,1),s) ) ;

son-positions( t.nil, xl, l-w, <t'.q,q>, l-w', d ) ->
  node-positions( t, x,  t', l-nw )
  min-distance( l-w, l-nw,  0, d )
  merge( l-w, shift(l-nw,d), l-w')
  freeze-on( xl, value( x, add(xl,d) ) ) ;
son-positions( t.l-t, xl, l-w, <t'.l-t',q>, l-w', d-max )->
  dif( l-t, nil )
  node-positions( t, x,  t', l-nw )
  min-distance( l-w, l-nw,  0, d )
  merge( l-w, shift(l-nw,d), l-wl )
  son-positions( l-t, xl, l-wl, <l-t',q>, l-w', d-max )
  freeze-on( xl, value( x, add(xl,d) ) ) ;

min-distance( nil, l-w, d, d ) -> ;
min-distance( l-w, nil, d, d ) -> ;
```

```
min-distance( (g1.r1).l1, (g2.r2).l2, d, d' ) ->
  no-overlap( r1, g2, d, d1 )
  min-distance( l1, l2, d1, d' ) ;

no-overlap( r1, g2, d, d' ) ->
  value( d', if( inf(add(r1,1),g2)
               , d
               , sub(add(r1,2),g2) ) ) ;

merge( l-w, shift(nil,d), l-w ) -> ;
merge( nil, shift((g2.r2).l2,d), (g.r).l ) ->
  value( g, add(g2,d) )
  value( r, add(r2,d) )
  merge( nil, shift(l2,d), l ) ;
merge( (g1.r1).l1, shift((g2.r2).l2,d), (g1.r).l ) ->
  value( r, add(r2,d) )
  merge( l1, l2, l ) ;

atom( t, ex(t), s ) -> ident( t )  length( t, s ) ;

...
tree-split( t1.t2, exm("."), 1, 2, t1.t2.nil ) -> ;
tree-split( <t1,t2>, ex(t1), s, 1, t2.nil ) ->
  ident( t1 )
  length( t1, s ) ;
...

margin( x, (g.r).l-w ) ->
  left-most( l-w, g, x ) ;

left-most( nil, g, g ) -> ;
left-most( (g0.r0).l-w, g, g'' ) ->
  value( g', if( inf(g0,g), g0, g ) )
  left-most( l-w, g', g'' ) ;

by-slice( nil ) -> ;
by-slice( l ) ->
  dif( l, nil )
  print-slice( l, <l-next-slice,nil> )
  new-line
  print-ramifications-of( l )
  new-line
  by-slice( l-next-slice ) ;

print-slice( nil, <q,q> ) ->;
print-slice( n.l-n, <l,q> ) ->
  print-node( n, <l,l1> )
  print-slice( l-n, <l1,q> ) ;

print-node( node(x,<o,g>), <q,q> ) ->
  blanks-until( sub(x,g) )
  o ;
print-node( node(x,<o,g,n,d-1>), d-1 ) ->
  blanks-until( sub(x,g) )
  o ;
```

```
print-ramifications-of( l ) ->
  { terminal dependant }

value( v, f ) -> val( f, v ) ;
```

4.0 CONCLUSION

If symbolic representation is the key for mental concept expression in A.I., graphical images are far easier to analyse for people. We have described some way to help to bridge the gap. The tools we have described will be included with the new Prolog II Marseille interpreter.

5.0 ACKNOWLEDGEMENT

I am indebted to A. Colmerauer for many fruitful discussions. He proposed the switch example.

BIBLIOGRAPHY

CLARK K.L. and TARNLUND S.A. 1977, "A First Order Theory of Data and Programs", pp. 939-944, IFIP Congr. Ser., Vol. 7, Publ: North-Holland, Amsterdam.

COLMERAUER A., KANOUI H. and VAN CANEGHEM M. 1981, "Last Steps Toward an Ultimate Prolog", Proc. 7th IJCAI, Vancouver.

COLMERAUER 1982, "Prolog and Infinite Trees", in: Logic programming, p231-251, A.P.I.C. Studies in Data Pocessing No.16, K.L.Clark and Tarnlund Ed., academic press, London.

PIQUE 1982, "On a Semantic Representation of Natural Language Sentences", Proceedings of the First International Logic Programming Conference, Sept. 14-17th 1982, pp.215-223, Marseille.

A Logical Reconstruction of Prolog II

M.H. van Emden

Department of Computer Science
University of Waterloo

J.W. Lloyd

Department of Computer Science
University of Melbourne

ABSTRACT

Colmerauer has proposed a theoretical model for Prolog II based on tree rewriting rather than logic. In this paper, we show that Prolog II can be regarded as a logic programming language.

1. Introduction

We take the view that a logic programming language is one in which a program is a first-order theory and computed answers are correct with respect to this theory (Clark 1979, Lloyd 1983).

One can then pose the question: is Prolog II (Colmerauer 1982a, Colmerauer 1982b) a logic programming language and, if so, in what sense is it? This question naturally arises from Colmerauer's account of his theoretical model for Prolog II. There, all explicit connection with first order logic has been severed. Instead, Prolog II is regarded as a system for rewriting possibly infinite trees. Unification is replaced by transformations on sets of equations.

Most Prolog implementations unify without occur check. This lack may lead to incorrect answers; hence it must be regarded as a shortcoming to be accepted for compelling reasons of execution efficiency. Prolog II also lacks the occur check. But Colmerauer considers this lack an essential feature of the language, accounting for it in his tree-rewriting model. Keeping in mind that the lack of occur check may lead to incorrect answers in ordinary Prolog, one may well ask whether Prolog II is a logic programming language.

We show that the answer to this question lies in making explicit Prolog II's theory of equality. Once that is done, it is easy to demonstrate that answers computed by Prolog II are correct with respect to a first-order theory consisting of (essentially) the program plus the equality theory.

Section 2 contains a brief account of Prolog II. In section 3, we introduce the idea of the "general procedure", which is an SLD-resolution proof procedure underlying both Prolog and Prolog II. In section 4 we show that Prolog is essentially the general procedure plus the equality theory $\{x = x\}$. (The meaning of "Prolog" here excludes any form of negation.) In section 5 Prolog II is shown to be essentially the general procedure plus a rather more complicated equality theory. What distinguishes Prolog from Prolog II then is the different way they handle equality. Section 6 contains some concluding remarks.

Throughout, P denotes a Horn-clause logic program not containing the predicate " = ". Similarly, G will always denote a goal which does not contain the predicate " = ".

2. Prolog II

The following brief description of Prolog II is taken from (Colmerauer 1982).

Definition An *equation* is an expression of the form $t_1 = t_2$ where t_1 and t_2 are terms.

Definition A set of equations is in *substitution* form if it is $\{x_1 = t_1, \ldots, x_n = t_n\}$, where $x_1, \ldots, x_n$ are distinct variables and none of $t_1, \ldots, t_n$ is a variable.

Definition A set $\{x_1 = t_1, \ldots, x_n = t_n\}$ of equations in *substitution* form *has a loop* if for some $k = 1,\ldots,n$, t_k has an occurrence of x_k or if such an occurrence of x_k can appear after possibly repeated substitutions in t_k using equations of the set.

In Prolog II, the solution of a set of equations is a substitution of trees for variables that makes both sides of each equation the same tree. A set of equations in substitution form is obviously solvable over the domain of rational trees. A set of equations in substitution form without a loop is obviously solvable over the domain of finite trees. Thus, equations can be solved by reducing them to substitution form by applying solution-preserving transformations.

Consider the following transformations (Colmerauer 1982):

Compaction:

> Eliminate any equation of the form $x = x$.

Variable Anteposition

> If x is a variable and t is not a variable, then replace $t = x$ by $x = t$.

Splitting

> Replace $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n)$ by $s_1 = t_1, \ldots, s_n = t_n$.

Confrontation

> If x is a variable and t_1, t_2 are not variables and the size of t_1 is not greater than the size of t_2, then replace $x = t_1, x = t_2$ by $x = t_1, t_1 = t_2$.

Variable Elimination

> If x and y are distinct variables, $x = y$ is in the system and x has other occurrences in that system, then replace these other occurrences of x by y.

He asserts that for any finite set of equations, application of the transformations in any order is only possible a finite number of times. Then either a set is obtained which is in substitution form or the set contains an equation of the form $t_1 = t_2$ where t_1 and t_2 have different outermost functions symbols. In the latter case the set has no solution over the domain of rational trees.

In Prolog II the clauses of a program are regarded as rules for rewriting a tree to a possibly empty sequence of trees. A query consists of a sequence of trees and a set of equations. A query is rewritten to another according to

$$< [A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_n], E > \rightarrow$$
$$< [A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n], E' >$$

if there is a rule

$$B \rightarrow B_1, \ldots, B_m \quad (m \geq 0)$$

in the program, if $E \cup \{B = A_i\}$ can be transformed to substitution form and if E' is such a form.

The final query in a derivation has an empty sequence of trees. The corresponding set of equations is the answer.

Now that we have given a brief overview of Prolog II, we are in a position to explain in what sense it is possible to give a logical reconstruction of Prolog II.

The domain of interest for Prolog II is the set of infinite trees. What we have to do is find a first-order theory for which the intended interpretation is a model

and also for which every answer computed by Prolog II is correct with respect to this theory. Naturally, the main part of this theory is the program itself. The remainder is simply a theory of equality. We have to find an equality theory so that each of the transformations employed by Prolog II (compaction, etc.) can be justified because they always produce a set of equations that is a logical consequence of the parent set of equations plus the equality theory.

3. The General Procedure

Definition The *homogeneous form* of a clause $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m$ is

$p(x_1, \ldots, x_n)$

$\leftarrow x_1 = t_1, \ldots, x_n = t_n, B_1, \ldots, B_m$

where $x_1, \ldots, x_n$ are distinct variables not appearing in the original clause.

Definition Let P be a program. The *homogeneous form* P' of P is the collection of homogeneous forms of each of its clauses.

Definition An atomic formula, whose predicate symbol is " = ", is called an *equation*.

We now describe the *general procedure*. We call it "general" because, depending on the theory of equality invoked after it, we get Prolog, Prolog II or other specialized languages.

The general procedure uses the homogeneous form P' of a program P and produces an SLD-derivation (Kowalski 1974, van Emden 1977). It consists of constructing, from some initial goal G, an SLD-derivation using input clauses from P', while never selecting an equation. The general procedure terminates if a goal consisting solely of equations is reached. Note that because of the homogeneous form of P' the general procedure never constructs bindings for the variables in the initial goal.

For a particular language, the general procedure needs to be supplemented by a theory E of equality. E is used to prove the equations resulting from the general procedure. During the proving of the equations, substitutions for the variables in the initial goal are produced. If the equation-solving process is successful (that is, the empty goal is eventually produced), then these substitutions for the variables in the initial goal are output as the answer.

The equation-solving process would normally be done by resolving goal clauses with clauses from the equality theory. However, other methods are possible. For example, the last step in the equation solving process for Prolog II is not a resolution step.

The introduction of the general procedure is purely a didactic device to explain which parts of Prolog and Prolog II are the same. Obviously, it would be very inefficient in practice since unsolvability of a set of equations is not detected until near the end of a computation. A practical system must perform some equation solving throughout a computation and, of course, this is what both Prolog and Prolog II do.

4. Equality theory for Prolog

Proposition 1. Let P be a program, G a goal and P' the homogeneous form of P. Then $P \cup \{G\}$ is unsatisfiable iff $P' \cup \{x = x\} \cup \{G\}$ is unsatisfiable.

Proof We first prove that P is a logical consequence of $P' \cup \{x = x\}$. Let M be a model for $P' \cup \{x = x\}$. We have to show M is a model for P. Take in P any clause $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m$ with variables $y_1, \ldots, y_d$. Suppose that for some assignment of these variables $B_1 \cdots B_m$ is true in M. Consider the homogeneous form

$p(x_1, \ldots, x_n)$

$\leftarrow x_1 = t_1, \ldots, x_n = t_n, B_1, \ldots, B_m$

of this clause in P'. Let x_i be the element assigned to t_i for the above assignments of the y_j's, for $i = 1,\ldots, n$. By the axiom $x = x$ and the assumption that $B_1 \cdots B_m$ is true in M, we have that $p(x_1, \ldots, x_n)$ is true in M. That is, $p(t_1, \ldots, t_n)$ is true in M. Consequently, M is a model for P and so P is a logical consequence of $P' \cup \{x = x\}$.

It follows from this that if $P \cup \{G\}$ is unsatisfiable, then so is $P' \cup \{x = x\} \cup \{G\}$.

Conversely, suppose $P' \cup \{x = x\} \cup \{G\}$ is unsatisfiable. Let M be a model for P. Then we can extend M to a model M' for $P' \cup \{x = x\}$ by assigning the identity relation to " = ". Thus G is false in M' and hence in M. (Note that G contains no occurrence of " = ".) Hence $P \cup \{G\}$ is unsatisfiable. □

Proposition 1 shows that the equality theory for Prolog is the single axiom $\forall x \; x = x$.

5. Equality theory for Prolog II

The equality theory E for Prolog II is rather more complex than the one for Prolog and consists of the following axioms:

1. $\forall x \; x = x$
2. $\forall x \forall y \;\; x = y \rightarrow y = x$
3. $\forall x \forall y \forall z \;\; x = y \;\; y = z \rightarrow x = z$
4. $\forall x_1 \cdots \forall x_n \forall y_1 \cdots \forall y_n$
   $(x_1 = y_1) \cdots (x_n = y_n)$
   $\rightarrow f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$,
   for all function symbols f.
5. $\exists x_1 \cdots \exists x_n \;\; \exists y_1 \cdots \exists y_k$
   $(x_1 = t_1) \cdots (x_n = t_n)$,
   where the x_i's are distinct variables, the t_i's are terms and $\{x_1, \ldots, x_n, y_1, \ldots, y_k\}$ is the set of all variables in the formula.

Note that axioms 4 and 5 are actually axiom schemas. The first task is to show that all the above axioms are true for the intended interpretation of " = " as the identity relation on the domain of infinite trees. Axioms 1 to 4 are the usual axioms for " = " and are certainly true in the intended interpretation. Axiom 5 is true by Colmerauer's solvable-form theorem (Colmerauer 1982). This theorem states that a system of equations $\{x_1 = t_1, \ldots, x_n = t_n\}$ has a solution in the domain of infinite trees, provided the x_i's are distinct variables.

Now we are in a position to prove our main result, which amounts to the soundness of Prolog II. Intuitively, it states that every answer computed by Prolog II is correct with respect to the first order theory consisting of the homogeneous form of the program plus the equality theory E.

Proposition 2. Let P be a program, P' its homogeneous form, G a goal and E the above equality theory for Prolog II. If Prolog II solves the goal G, then $P' \cup E \cup \{G\}$ is unsatisfiable.

Proof Since the general procedure uses resolution, it produces intermediate goals all of which are a logical consequence of $P' \cup \{G\}$. We now verify that each of the five transformations of Prolog II can be justified on the basis of resolution steps using the equality theory E.

Compaction

Consider a goal $\leftarrow y = y, e_1, \ldots, e_k$, where $e_1, \ldots, e_k$ are equations. Elimination of $y = y$ is justified by resolving the goal with the equality axiom $\forall x \; x = x$. Thus $\leftarrow e_1, \ldots, e_k$ is a logical consequence of $\{\leftarrow y = y, e_1, \ldots, e_k\} \cup E$.

Variable Anteposition

This is justified in a similar way to compaction, but using axiom 2.

Splitting

Resolve with axiom 4.

Confrontation

It suffices to show that $\leftarrow x = t_1,\ t_1 = t_2$ is a logical consequence of $\{\leftarrow x = t_1, x = t_2\} \cup E$. Indeed we have the following derivation:

$\leftarrow x = t_1, x = t_2$

$\leftarrow x = t_1, x = t_1, t_1 = t_2$

(resolving with an instance of axiom 3)

$\leftarrow x = t_1, t_1 = t_2$

Variable elimination

We let $s[x/y]$ denote the result of replacing in s all occurrences of x (if any) by y. The following lemma will be useful.

Lemma

$x = y \rightarrow s = s[x/y]$

and $x = y \rightarrow s[x/y] = s$

are logical consequences of E.

The proof is by repeated applications of axioms 1 and 4, plus an application of axiom 2.

To justify variable elimination, it suffices to show that

$\leftarrow x = y, s[x/y] = t[x/y]$

is a logical consequence of

$\{\leftarrow x = y, s = t\} \cup E$.

Indeed we have the following derivation:

$\leftarrow x = y,\ s = t$

$\leftarrow x = y,\ s = s[x/y],\ s[x/y] = t$ (axiom 3)

$\leftarrow x = y,\ x = y,\ s[x/y] = t$ (lemma)

$\leftarrow x = y,\ s[x/y] = t$

$\leftarrow x = y,\ s[x/y] = t[x/y],\ t[x/y] = t$

(axiom 3)

$\leftarrow x = y,\ s[x/y] = t[x/y],\ x = y$ (lemma)

$\leftarrow x = y,\ s[x/y] = t[x/y]$.

Finally, the last step in a Prolog II computation is the application of the solvable form theorem. From a logical point of view, this is equivalent to an application of axiom 5 above.

This completes the proof of the proposition. □

6. Concluding Remarks

In (Colmerauer 1982) the theoretical model of Prolog II is extended to cope with inequalities. We have not attempted to deal with these.

Note that the general procedure can be followed by the use of any theory of equality. We have given two useful theories in this paper. It should be interesting to consider other equality theories. We are particularly interested in theories suggested by two existing systems related to Prolog. The first is DLOG (Goebel 1984) logic-based database management system which uses two different equality theories: one for equality of descriptions and the other for heuristic evaluation of queries. The second is a version of Prolog (Kornfeld 1983) with an extended unification.

7. Acknowledgments

Many thanks to Sten-Ake Tarnlund for his suggestions for improvement.

We are indebted to Imperial College of the University of London for its hospitality which made this work possible. MHvE gratefully acknowledges support from the UK Science and Engineering Research Council. We also acknowledge our debt to the Canadian National Science and Engineering Research Council for providing document preparation facilities.

8. References

K.L. Clark: Predicate logic as a computational formalism. Research Report 79/59, Department of Computing, Imperial College, 1979.

A. Colmerauer: Prolog and infinite trees. pp. 231-251 in: K.L. Clark and S.A. Tarnlund (eds.): Logic Programming. Academic Press, 1982 (a).

A. Colmerauer et al.: Prolog II Reference Manual and Theoretical Model. Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille, October 1982 (b).

M.H. van Emden: Programming with resolution logic. pp. 266-299 in: E.W. Elcock and D. Michie (eds.): Machine Intelligence 8. Ellis Horwood, 1977.

R.G. Goebel: A logic data model for the machine representation of knowledge. PhD Dissertation, Dept. of Computer Science, University of British Columbia, 1984.

Hansson, A. and Haridi, S. Programming in a natural deduction framework. Proc. Symp. Functional Languages and Computer Architecture, Gothenburg 1981.

Hansson, A., Haridi, S. and Tarnlund, S-A. Properties of a logic programming language. In: Clark, K.L. and Tarnlund, S-A. Logic Programming, Academic Press, 1982.

W.A. Kornfeld: Equality for Prolog. Proc. 8th International Joint Conference on Artificial Intelligence. A. Bundy (ed.), William Kaufmann, Los Altos, 1983.

R.A. Kowalski: Predicate logic as a programming language. Proc. IFIP74, pp. 569-574.

J.W. Lloyd: Foundations of logic programming. Technical Report 82/7, University of Melbourne, revised May 1983.

A COMPARISION OF TWO LOGIC PROGRAMMING LANGUAGES: A CASE STUDY

Szőts Miklós
Research Institute of Applied Computer Science
P.O.Box 146 Budapest 112
H 1502 Hungary

ABSTRACT

Two logic programming languges, the well known PROLOG and the new LOBO are compared. LOBO is defined. Two examples dealing with planar covering problems are analyzed. It is shown that both languages are able to realize the same algorithms. However LOBO is nearer to traditional languages: it does not use pattern matching, it can be complied easily, and it is able to use traditional features of programming.

INTRODUCTION

A comparision of two logic programming languages, namely PROLOG and LOBO (defined here) is presented. The two languages studied here are equivalent in the sense that both are suitable to define every partial recursive functions. In this sense both can be considered universal. The question is, what class of algorithms can be realised in them. Here theoretical comparison is not presented, a forthcoming paper will do it, but a case study is analyzed. Programs are introduced dealing with planar covering problem, namely how a rectangle can be covered by given elements.

DEFINITION OF LOBO

All formulas we write down belong to the language of arithmetics of integers, that is they belong to the language whose similarity type includes the numerals as constants, function symbols $+,-,\cdot,\underline{div},\underline{rem}$, and so on, relation symbols $<,\leqslant$, and so on. Let $\mathfrak{I}$ denote the standard model of integers, and let Ax be an axiom system of integer arithmetics. Clearly, $\mathfrak{I}$ is the only Herbrand interpretation in Mod (Ax).

Two sets of formulas are defined:

$$\Phi=\{\tau_1<y\leqslant\tau_2:\tau_1 \text{ and } \tau_2 \text{ are term free of } y\}\cup\{\tau=y:\tau \text{ is a term free of } y\}$$

$$\Phi_q=\{\tau<y:\tau \text{ is a term free of } y\}$$

In the following formulas belonging to Φ and Φ_q are used to bound the domain of bound variables. Quantifiers in the form $\forall y(\varphi(y)\rightarrow\psi(y))$, if $\varphi(y)$ belongs to Φ, and $\exists y(\varphi(y)\wedge\psi(y))$, if $\varphi(y)$ belongs to Φ or Φ_q, are called bounded quantifiers. We define a language, where all quantifiers are bounded.

Definition 1

C is said to be the set of cuttable formulas and is defined inductively as follows:

(i) quantifier-free formulas belong to C;

(ii) if ψ_1 and ψ_2 belong to C, then formulas $\psi_1\wedge\psi_2$, $\psi_1\vee\psi_2$ also belong to C;

(iii) if ψ belongs to C, then, formula $\neg\psi$ also belongs to C;

(iv) if $\varphi(y)\in\Phi$, and ψ belongs to C, then $y(\varphi(y)\rightarrow\psi)$ also belongs to C;

(v) if $\varphi(y)\in\Phi$, and C belongs to C, then $y(\varphi(y)\wedge\psi)$ also belongs to C;

(vi) only formulas obtained by the above rules belong to C.

Definition 2

C_q is said to the set of quasi-cuttable formulas and is defined inductively as follows:

(i) quantifier-free formulas belong to C_q;

(ii) if ψ_1 and ψ_2 belong to C_q then formulas $\psi_1\wedge\psi_2$, $\psi_1\vee\psi_2$ also belong to C_q;

(iii) if ψ belongs to C then formula $\neg\psi$ belongs to C_q;

(iv) if $\varphi(y)\in\Phi$ and ψ belongs to C_q, then $\forall y(\varphi(y)\rightarrow\psi)$ also belongs to C_q;

(v) if $\varphi(y)\in\Phi\cup\Phi_q$ and ψ belongs C'_q, then $\exists y(\varphi(y)\wedge\psi)$ also belongs to C_q;

(vi) only formulas obtained by the above rules belong to C_q.

Definition 3

The languages $\langle C_q,\{I\},\models\rangle$ is called the Language of Quasicuttable Formulas (LQF).

Every cuttable or quasi-cuttable formula has a well defined truthvalue in I. A calculus is presented to determine this truthvalue.

We give this rules in the form of algol-like programs. Let p denote a function with C_q as domain, and the set of algol like programs as range. If $\psi_i(\bar{x})$ is a quasicuttable formula, $p(\psi_i)$ is a procedure to compute the truthvalue of $\psi_i(\bar{a})$, where $\bar{a}$ is an evaluation of $\bar{x}$, as an input for procedure $p(\psi_i)$.

Table 1.

	ψ_i	$p(\psi_i)$
1.	quantifier-free formula of type t_i	$z_i \leftarrow \psi_i$;
2.	$\psi_j \vee \psi_k$	$p(\psi_j)$ if z_j then $z_i \leftarrow$ true else begin $p(\psi_k)$ $z_i \leftarrow z_k$; end;
3.	$\psi_j \wedge \psi_k$	$p(\psi_j)$ if z_j then begin $p(\psi_k)$ $z_i \leftarrow z_k$; end else $z_i \leftarrow$ false;
4.	$\neg\psi_j$	$p(\psi_j)$ $z_i \leftarrow \neg z_j$;

ψ_i	$p(\psi_i)$

1. $\forall y((\tau_1<y_i\leqslant\tau_2)\rightarrow\psi_j)$	$(y_i,z_i) \leftarrow (\tau_1,$ <u>true</u>$)$;
	<u>while</u> $z_i\wedge(y<\tau_2)$ <u>do</u>
	  <u>begin</u>
	  $y_i \leftarrow y_i+1$;
	  $p(\psi_j)$
	  $z_i \leftarrow z_j$;
	<u>end</u>;

2. $\exists y_i((\tau_1<y_i\leqslant\tau_2)\wedge\psi_j)$	$(y_i,z_i) \leftarrow (\tau_1,$ <u>false</u>$)$;
	<u>while</u> $\neg z_i\wedge(y<\tau_2)$ <u>do</u>
	  <u>begin</u>
	  $y_i \leftarrow y_i+1$;
	  $p(\psi_j)$
	  $z_i \leftarrow z_j$;
	<u>end</u>;

3. $\exists y_i(y_i=\tau)\wedge\psi_j$	$y_i \leftarrow \tau$; $p(\psi_j)$ $z_i \leftarrow z_j$

4. $\exists y_i(\tau<y_i)\wedge\psi_j$	$(y_i,z_i) \leftarrow (\tau,$ <u>false</u>$)$;
	<u>while</u> $\neg z_i$ <u>do</u>
	  <u>begin</u>
	  $y_i \leftarrow y_i+1$;
	  $p(\psi_j)$
	  $z_i \leftarrow z_j$;
	<u>end</u>;

5. $R(\tau_o,\ldots,\tau_{n-1})$	<u>call</u> $R(\tau_o,\ldots,\tau_{n-1}, z_i)$;

6. $\psi_j(y_i)[y_i/F(\tau_o,\ldots$
$\ldots,\tau_{n-1})]$
or
$\exists y_i(y_i=F(\tau_o,\ldots,\tau_{n-1})\wedge$
$\wedge\psi_j(y_i)$
	<u>call</u> $F(\tau_o,\ldots,\tau_{n-1}, y_i, z_F)$;
	<u>if</u> z_F <u>then</u>
	  <u>begin</u>
	  $p(\psi_j)$
	  $z_i \leftarrow z_j$;
	  <u>end else</u> $z_i \leftarrow$ <u>false</u>;

Table 2.

Boolean variable z_i in $p(\phi_i)$ gets the truthvalue.

The definition of mapping p for quasi-cuttable formulas can be found in Table 1 and in rows 1-4 of Table 2. The calculus consisting of these rules let be denoted by K.

Theorem 1

Let $\phi(x_o,\ldots,x_{n-1})$ be a quasi-cuttable formula with free variables $x_o,\ldots,x_{n-1}$. If $I \models \phi(x_o,\ldots,x_{n-1})[a_o,\ldots,a_{n-1}]$, then $K \vdash \phi[x_o/a_o,\ldots,x_{n-1}/a_{n-1}]$.

In the following we deal with definitions rather then theorems. Let R and F be relation and function symbols not occuring in the similarity type in question. Definitions of relation R and function F respectively are formulas

$$R(x_o,\ldots,x_{n-1}) \leftrightarrow \rho(x_o,\ldots,x_{n-1}) \quad \text{and}$$

$$F(x_o,\ldots,x_{n-1})=y \leftrightarrow \rho(x_o,\ldots,x_{n-1},y).$$

If the new symbol occurs in formula ρ (defining formula), the definition is called implicit, if not, it is called explicit. In the case of definition of a function, let $\varphi(x_o,\ldots,x_{n-1})$ be a quasi-cuttable formula having variable y as an existentially bound variable. If $\varphi(\bar{x})$ is equivalent with $\exists y \rho(\bar{x},y)$, $\varphi(x)$ is considered as the quasi-cuttable definition of $F(\bar{x})$, and it is denoted by $F(x) \stackrel{d}{=} \varphi(x)$. The most important properties of definitions with quasi-cuttable defining formulas are stated by the following theorems.

Theorem 2

Every partial recursive functions can be defined by quasi-cuttable formulas.

Theorem 3

Definition with quasi-cuttable defining formula has effectively computable least fix point.

The question, we are interested in, is whether a given n-tulpe belongs to a relation defined by a quasi-cuttable formula. The case of explicit definitions is covered by Theorem 1. To handle implicit definitions new inference rules are introduced. The corresponding program segments can be seen in the rows 5,6 of Table 2. Calculus K completed with the new rules is denoted by K_d.

Definition 4

The pair $\langle LQF, K_d \rangle$ is called the LOgic of BOunded quantifiers (LOBO).

Theorem 4

Calculus K_d is a complete calculus for unfolding quasi-cuttable definitions that is, if $\langle a_o,\ldots,a_{n-1} \rangle$ belongs to the least fix point of the definition $R(\bar{x}) \leftrightarrow \rho(\bar{x})$, then $\rho(\bar{x})[x_o/a_o,\ldots,x_{n-1}/a_{n-1}]$ can be proved in K_d.

LOBO can be considered as a logic programming language. Quasi-cuttable formulas are programs, with free variables as input variables, and existentially bound variables as output ones. A program realizing function p computes a program written in a traditional programming language from any quasi-cuttable formula.

Here we only outlined the most important facts, for further details see [1],[2],[3].

THE EXAMPLES

Two problems are presented, both dealing with covering a rectangle with given elements. The first problem is a special task coming from architectural CAD, the second one can be considered general.

1. The special problem

A rectangle is given with edges of length X,Y respectively. It has to be covered by rectangular elements of given measurements. Elements of unlimited numbers can be used from each type. However, the covering has to meet the following requirements:

(i) the same element must be applied in all 4 corners;
(ii) apart from the corner elements, the same element must be applied along the edges in the x-direction;
(iii) apart from the corner elements, the same element must be applied along the edges in the y-direction;
(iv) apart from the corner and edge elements, the same element must be applied troughout the remainder of the rectangle.

Let us see first the PROLOG program.

A remark: the MPROLOG syntax is used (see [4]), but MPROLOG pecularities are avoided. Constants are written with lower, and variables with upper case letters.

The types of elements are represented in facts of the form element(P,X1,Y1), where P is an identifier of the type, X1 and Y1 are the length of edges in x and y direction respectively. The rule describing the possible coverings can be formularized in one clause:

```
covering(X,Y,P1,P2,P3,P4) ←
    element(P1,X1,Y1),
    element(P2,X2,Y1),
    rem(X-2·X1,X2,Ø),
    element(P3,X1,Y2),
    rem(Y-2·Y1,Y2,Ø),
    element(P4,X2,Y2).        (PP1)
```

Let us complete the above clause with facts:

```
element(first,4,2).           (PP2)
element(second,3,2).          (PP3)
element(third,2,2).           (PP4)
element(fourth,2,1).          (PP5)
```

and with goal statement ← covering (13,8,P1,P2,P3,P4).
The search tree and the solution is shown on Figure 1.

In LOBO programs the bounding formulas are written as upper indeces of the corresponding quantifier, and the corresponding connective ($\wedge$ or $\rightarrow$) is omitted.

The LOBO program to solve our problem can be seen in Figure 2.

Here N is the numer of types of elements, and the measurements of the i-th type are stored in the i-th row of in a two dimensional array ELEMENT. In the same concrete case as above, N is 4 and array ELEMENT is:

```
ELEMENT(1,1)=4, ELEMENT(1,2)=2,
ELEMENT(2,1)=3, ELEMENT(2,2)=2,
ELEMENT(3,1)=2, ELEMENT(3,2)=2,
ELEMENT(4,1)=2, ELEMENT(4,2)=1.
```

The and/or tree representing the proof search of the formula can be seen on Figure 3. Clearly the search tree of the PROLOG and the LOBO program is basically the same.

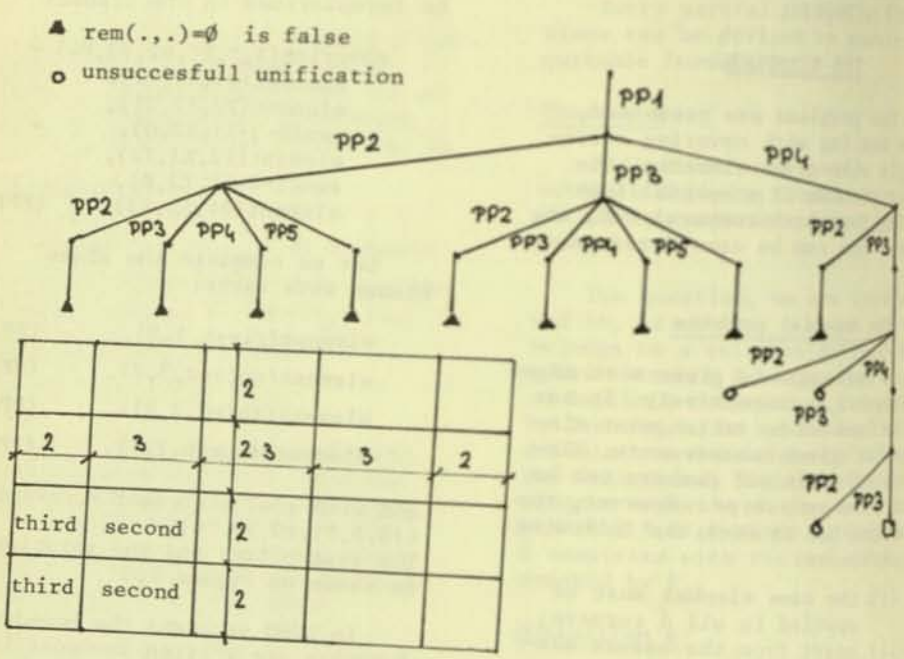


Figure 1.

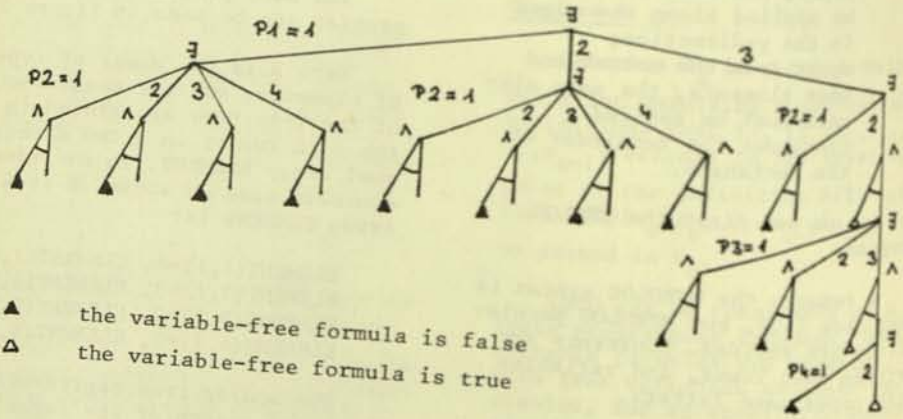
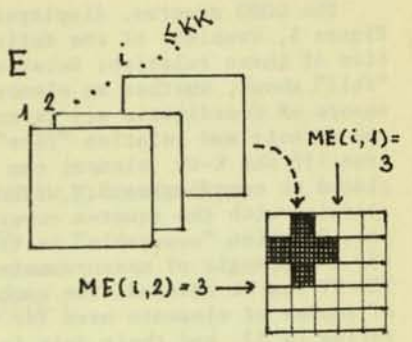


Figure 3.

2. The general problem

The rectangle to be covered is supplied with a mesh, whose parallels are at equal distance. So the rectangle can be considered consisting of elementary squares. The elements, which can be used in covering, are also considered being constructed from such elementary squares. The elements cannot be rotated or reflected. We keep on supposing that from each type there is an unlimited supply.

First let us see the LOBO program. There are KK types. These types are represnted by two arrays. Array E is of three dimensions. Values E(i,.,.) describe the i-th type by giving the characteristic function of the element: E(i,j1, j2) is different from Ø iff the elementary square with coordinates j1,j2 is covered by the i-th element. Every element can be covered by a rectangle, the length of its edges are in ME(i,1) and ME(i,2). The foregoing conditions are illustrated on Figure 4.



E(i,1,2),E(i,2,1),E(i,2,2),
E(i,2,3), E(i,3,2)

are not Ø, other E(i,k,ℓ)
values are Ø

Figure 4.

```
covering(X,Y,ELEMENT,N)≡d
 ∃ P1^{Ø<P1≤N} ∃ P2^{Ø<P2≤N}
     {(ELEMENT(P1,2)=ELEMENT(P2,2)∧
       rem(X-2·ELEMENT(P1,1),ELEMENT(P2,1))=Ø)∧
       ∃ P3^{Ø<P3≤ N}
          {(ELEMENT(P1,1)=ELEMENT(P3,1)∧
            rem(X-2·ELEMENT(P1,2),ELEMENT(P3,2))=Ø)∧
            ∃ P4^{Ø<P4≤N}
               (ELEMENT(P2,1)=ELEMENT(P4,1)
                ELEMENT(P2,2)=ELEMENT(P4,2))}}
```

Figure 2.

The LOBO program, displayed on Figure 5, consists of the definition of three relation. Relation "full" shows, whether an elementary square of coordinates x,y is covered or not; and relation "fits" is true iff the K-th element can be placed at coordinates X,Y without conflict with the squares covered yet. Relation "coverable" is true iff a rectangle of measurements XX, YY can be covered. The numbers of copies of elements used for covering is JJ, and their data is expressed by function RESULT as follows.

RESULT(I,1) is the serial number of the I-th covering element; RESULT(I,2), and RESULT(I,3) show its position (see Figure 6).

Note that formulas $\forall i \exists j \psi(j)$ and $\exists f \forall i(\psi(j) \lceil i/f(j) \rceil)$ are equivalent. The only "dirty" trick in the program is that variable symbol j is substituted by f(j), - that is NEWRESULT(JJ,1), and so on, - and this expression is not handled as a term but as a variable. However this notation helps to represent this function by an array in the program. Logical purity makes us to use a different function symbol (NEWRESULT) when new values are defined. However when the formula is transformed to a program, the same array identifier can be used for RESULT and NEWRESULT.

To interpret variables X,Y,H,I in the defining formula of relations coverable properly, see Figure 6.

Using PROLOG, the types of elements are characterized by facts, as Figure 7 shows. The representation of the partially covered rectangle also occurs using facts: for each pair of coordinates X,Y there is a statement full(X,Y) or empty(X,Y), depending whether the corresponding square is covered or not.

```
e(cross,3,3,<1,2>.<2,1>.
 <2,2>.<2,3>.<3,2>.nill).
```

Figure 7.

The program:

```
coverable(XX,YY)←cover(XX,YY).
coverable(XX,YY)←covered.

cover(XX,YY)←empty(X,Y),
   e(K,N,M,LIST), at(LIST,H,I),
   X1 is X-H, Y1 is Y-I,
   X1+N≤XX, Y1+M≤YY,
   fits (K,LIST,X1,Y1),
   cover(XX,YY).

at(<H,I>.LIST,H,I).
at(<G,F>.LIST,H,I)←at(LIST,H,I).

fits(K,nill,X,Y).
fits(K,<G,F>.LIST,X,Y) ←
   X1 is X+G,Y1 is Y+F,
   empty(K,X1,Y1), modify(K,X1,Y1),
   fits(K,LIST,X,Y).

modify(K,X1,Y1)←
   fsupclause(empty(X1,Y1)),
   assclause(full(K,X1,Y1)).

covered←not(empty(X1,Y1)).
```

Here fsupclause and assclause are built-in predicates, the first deletes, the second creates a clause. Both are backtrackable.

Note that the PROLOG program follows the structure of the LOBO program. Significant difference is only in representation of the par-

$$
\begin{aligned}
&\text{coverable}(XX,YY,E,ME,KK,JJ,RESULT) \stackrel{d}{\equiv} \\
&[\exists X^{\emptyset<X\leq XX}\exists Y^{\emptyset<Y\leq YY}\ \neg\text{full}(E,JJ,RESULT,X,Y)\wedge \\
&\quad [\exists K^{\emptyset<K\leq KK}\exists H^{\emptyset<H\leq ME(K,1)}\exists I^{\emptyset<I\leq ME(K,2)} \\
&\qquad ((X-H+ME(KE,1))\leq XX)\wedge(Y-I+ME(K,2)\leq YY))\wedge \\
&\qquad \text{fits}(E,JJ,RESUTL,X-H,Y-I)\wedge \\
&\qquad \forall L^{0<L\leq JJ}\forall N^{0<1\leq 3}\exists \text{NEWRESULT}(L,N)^{\text{NEWRESULT}(L,N)=\text{RESULT}(L,N)} \\
&\qquad\quad \exists \text{NEWRESULT}(JJ+1,1)^{\text{NEWRESULT}(JJ+1)=K} \\
&\qquad\quad \exists \text{NEWRESULT}(JJ+1,2)^{\text{NEWRESULT}(JJ+2)=X-H} \\
&\qquad\quad \exists \text{NEWRESULT}(JJ+1,3)^{\text{NEWRESULT}(JJ+1,3)=Y-I} \\
&\qquad\qquad \text{coverable}(XX,YY,E,ME,KK,JJ+1,\ \text{NEWRESULT})]]\vee \\
&[\forall X^{\emptyset<X\leq XX}\forall Y^{\emptyset<Y\leq YY}\ \text{full}(E,JJ,RESULT,X,Y)]
\end{aligned}
$$

$$
\begin{aligned}
&\text{full}(E,JJ,RESULT,X,Y) \stackrel{d}{\equiv} \\
&\exists J^{\emptyset<J\leq JJ}[\emptyset<X-RESULT(J,2)\leq ME(RESULT(J,1),1)\wedge \\
&\qquad \emptyset<Y-RESULT(J,3)\leq ME(RESULT(J,1),2)\wedge \\
&\qquad E(RESULT(J,1),X-RESULT(J,2),Y-RESULT(J,3))\neq\emptyset]
\end{aligned}
$$

$$
\begin{aligned}
&\text{fits}(K,E,ME,JJ,RESULT,X,Y) \stackrel{d}{\equiv} \\
&\forall H^{\emptyset<H\leq ME(K,1)}\forall I^{\emptyset<I\leq ME(K,2)} \\
&\quad E(K,H,I) \rightarrow \neg\text{full}(E,J,RESULT,X+H,Y+I)
\end{aligned}
$$

Figure 5.

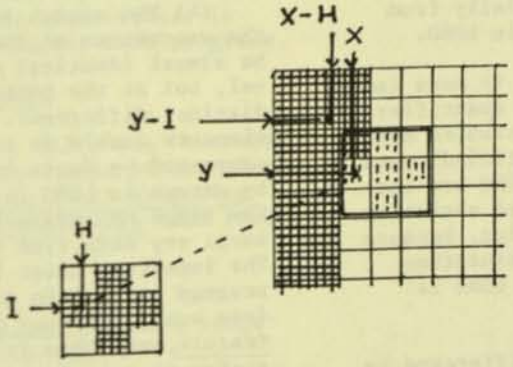


Figure 6.

tially covered rectangle.

COMPARISON

The two examples presented above show that the algorithms which can be described by the two languages may be the same. The analysis of equivalences and differences will show that this experience can be generalized. There are few essential differences beetwen the two languages in spite of the different syntax and calculi.

(1) The basic difference beetwen the two languages is that PROLOG is based on a strict normal form making superflous most connectives, while in LOBO one uses all the logical connectives and quantifiers. In the case of connectives it is not an important difference. MPROLOG syntax allows to use connective "or" in the antecendent of a clause. As an example, the partition "coverable" may be written as:

```
coverable(XX,YY)←cover(XX,YY);
                 covered.
```

where ";" reads ∨. The usual interpretation of "not" in PROLOG does not differ essentially from the usage of negation in LOBO.

(2) The difference is more important in the case of quantifiers. In PROLOG rules all variables are free (universally quantified) ones. Their equivalents in LOBO are the existentially quantified variables. This is not contradiction, because PROLOG is based on a refutation proof procedure, while LOBO is based on a direct one.

The most evident difference is the universal quantifier in LOBO. If connective "not" can be used in the PROLOG version in question, sometimes "∀x" can be substituted by "¬∃x¬", as it was done in the definition of "covered". Otherwise the partition corresponding to a subformula begining with universal quantifier has to be programmed on a roundabout way.

(3) The interesting point is that the almost identical search trees of PROLOG and of LOBO programs are organized by different tools. In the case of LOBO, loops running on the bounded variables are explicitly expressed by the bounding formulas. In the case of PROLOG, the search strategy controls loops on the different clauses in the same partition, using first of all the pattern matching mechanism. It is a bargain: LOBO looses the possibilities provided by the pattern matching mechanism, - that is the role of input and output variables are fixed, and equality has to be stated explicitly. However, LOBO gaines the possibility of simple implementation, moreover of simple compilation. Note that this basic difference is difference in the calculi. The difference is the syntax occurs, because syntax in both cases follow the demands of calculi.

(4) The search trees that is the executions of the programs may be almost identical at the top level, but at the bottom there is a distinct difference. The data of elements usable in covering are expressed by facts in PROLOG, and by arrays in LOBO in both examples. The usage of arrays is not compulsory: any data type can be used. The important point is that a LOBO program can use in a quantifier free subformula any programming feature, available at the computer system in question. In PROLOG the built-in predicates correspond to this feature, but they are provided in a limited supply.

The importance of this factor can be seen in the second example. While the LOBO program is expressed in "pure logic", the PROLOG one is based on such metalogical features as the built-in predicates rewriting the formula itself. Without this possibility the equivalent of array RESULT has to be a list structure overburdening the program by handling lists.

SHORT SUMMARY

PROLOG and LOBO seem to be basicly equivalent logic programming languages. The most important differences:

(i) LOBO does not use pattern matching, loosing so some programming facilities, and gaining the possibility of simple compilation.

(ii) LOBO can use any progamming feature of the computer system.

These differences makes us claim that LOBO is nearer to traditional programming. However we think that it can play an important role in developing fifth generation computer systems. In [1] a simple non-von-Neumann architecture is suggested to execute LOBO programs.

REFERENCES

[1] Gergely T., Szőts M.: "Cuttable formulas for logic programming" 1984 International Symposium on Logic Programming IEEE Press, 1984.

[2] Gergely T., Szőts M.: "LOBO - a new logic programming language" in preparation

[3] Szőts M., Csizmazia S.: "A method for program synthesis" International Symposium on Programming 5th Colloquium Lecture Notes in Computer Science 137 Springer Verlag 1982.

[4] "MPROLOG language reference manual" SZKI, Budapest 1982.

COMPUTATION TREES AND TRANSFORMATIONS OF LOGIC PROGRAMS

Olga Štěpánková
Institute for Computation Techniques, ČVUT
Horská 3, 128 00 Praha 2
Czechoslovakia

Petr Štěpánek
Charles University
Malostranské náměstí 25
118 00 Praha 1
Czechoslovakia

ABSTRACT

We shall introduce new concept of computation trees of logic programs and we shall use it in reasoning about programs. We shall describe three types of transformations improving the structure of logic programs. There are two natural measures of complexity suggested by computation trees, namely, the number of nodes called by recursion and the maximal number of AND/OR alternations on a branch. We shall show that both measures collapse, more precisely, we shall show that every logic program can be transformed to a program computing the same function the computation tree of which has at most one called node and at most two alternations on every branch. We shall discuss some conclusions related to this Normal Form Theorem.

0 INTRODUCTION

Problem reduction based on decomposition of goals to several subgoals is a prominent feature of the procedural interpretation of Horn Logic used in Logic programming. It is well-known that problem reduction can be naturally depicted by AND/OR graphs with alternating and- and or-nodes. D. Harel /1980a, 1980b/ described a simple tree-like programming specification language of so called AND/OR-schemes which allow to capture the logical structure of programs developed by the stepwise synthesis in the discipline of structured programming. It was shown in /Štěpánková et al. 1983/ that AND/OR schemes are naturally embedded in the class of logic programs, namely, that to every AND/OR-scheme corresponds a logic program computing the same relation. There are logic programs, however, which cannot be described by an AND/OR-scheme.

In this paper, we shall introduce a new concept of computation trees for logic programs extending the definition of a computation tree from /Šebelík et al. 1982/. The extension is motivated by AND/OR-schemes.

We shall describe three types of transformations of computation trees which allow us

- to avoid recursion calls from one branch of the tree to another
- to move the nodes called by clean recursion closer to the root
- to push upwards the nodes of OR-branching

These transformations have many interesting implications to logic programs. One of them is the existence of a Normal Form of Logic Programs. This generalizes a similar result due to /Harel, 1980b/ concerning AND/OR-schemes.

The computation tree of every program in normal form has the following properties

/i/ there are at most two alternations of AND- and OR-nodes on every branch

/ii/ there is exactly one node to which refers every recursion call

We suppose that the reader is familiar with the operational and least fixed-point semantics of logic programs introduced in /van Emden and Kowalski 1976/.

1 COMPUTATION TREES

We shall use the standard graph-theoretic concepts like node, edge, leaf, root and branch. If we describe a tree, we usually put the root on top, the branches growing down. Hence the only parent node is above and all the successors of a node are below it. We speak about the depth of a node instead of its height. We call a node internal if it is not a leaf.

Let L be a first-order language and R be a predicate in L. An AND/OR-tree T is called a computation tree for R provided that it has the following properties

/i/ the root of T is an OR-node labelled by $R/v_1,\ldots,v_n/$ where $v_1,\ldots,v_n$ is an appropriate tuple of distinct variables. Every OR-node of T is labelled by an atomic formula of L and the labels of internal OR-nodes consist of a predicate symbol and a tuple of distinct variables.

/ii/ If n is an OR-node with the label A, all its successors are AND-nodes labelled by Horn clauses the head of which contains the same predicate symbol as A. Every edge connecting n with its successor n' is labelled by a substitution which unifies A with the head of the label of n'.

/iii/ To avoid multiplicity in defining predicates, the labels of different internal OR-nodes have different predicate symbols.

/iv/ If n is an AND-node labelled by the clause

$$B \leftarrow A_1,\ldots,A_k$$

then for every $i \leq k$, there is a successor OR-node n_i of n the label of which contains the same predicate symbol as A_i.

Moreover, if n_i is a leaf, its label is A_i. If n_i is internal and A_i is of the form $Q/t_1,\ldots,t_m/$, where $t_1,\ldots,t_m$ are terms, the label of n_i is $Q/u_1,\ldots,u_m/$, where u_j are distinct new variables and there are other successor leaves of n_i labelled by the equalities $u_j = t_j$ for $j \leq m$. If a term t_j

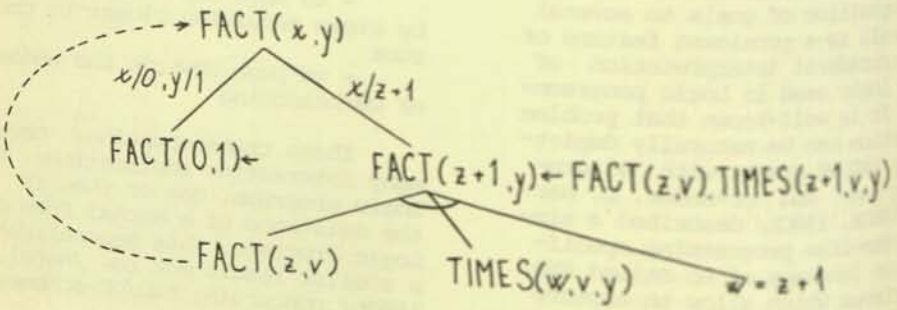


Figure 1

is a variable, we can use it instead of u_j and leave out the successor labelled $u_j = t_j$.

/v/ Every AND-node labelled by an unconditional statement B is a list.

Example 1. Let L be the language of arithmetic containing two constants 0 , 1 denoting zero and one and a binary function + for addition of natural numbers. The computation tree for the factorial of x is on Figure 1.

We distinguish two types of OR-leaves according to the attached predicate symbols. We call the leaf primitive if its predicate symbol is different from every predicate attached to an internal OR-node, otherwise we say that it is a call-leaf. Since the predicate symbols attached to internal OR-nodes are different, the predicate symbol attached to a call-leaf 1 coincides with the predicate symbol of exatly one internal OR-node, which is a called node /called by 1/. Note that the computation tree from Figure 1 contains only one call leaf and one called node. They are connected by a dashed bow.

It follows from the definition that the set of all clauses labelling the AND-nodes of a computation tree for the predicate R is a logic program computing R. On the other hand, if $\mathbb{P}$ is a logic program computing R , it is not difficult to construct a computation tree for R which corresponds to the program $\mathbb{P}$. If there is a recursion in $\mathbb{P}$, there might be several OR-nodes with the same attached predicate symbol. Thus we have to decide which of these nodes will be internal, the remaining ones being leaves. Hence there may be finitely many computation trees for a predicate R corresponding to a given program $\mathbb{P}$.

2 COMPUTATION TREES AND TIDY PROGRAMS

Let $\mathbb{A}$, $\mathbb{B}$ be logic programs. We say that $\mathbb{A}$ extends $\mathbb{B}$ iff the denotation /see van Emden and Kowalski 1976/ of any predicate P of $\mathbb{B}$ in $\mathbb{A}$ is the same as that in $\mathbb{B}$, i.e.

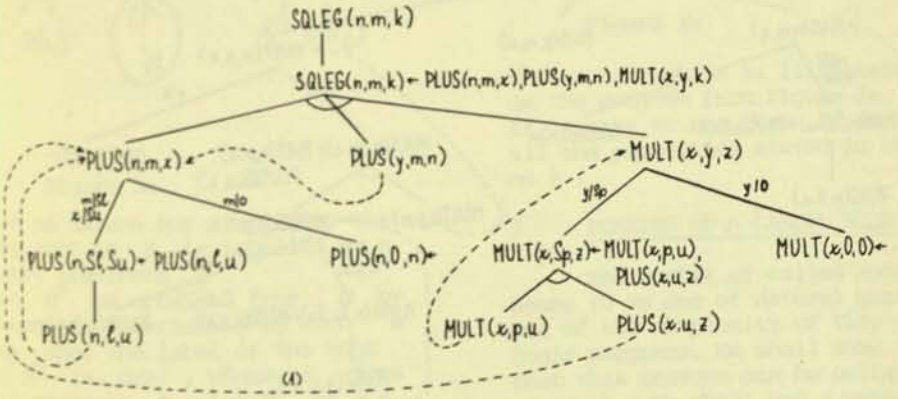


Figure 2a SQLEG/n,m,k/ iff $k = (n + m)(n - m)$

$$\{(t_1,\ldots,t_k):\ \mathbb{A} \vdash P(t_1,\ldots,t_k)\} =$$

$$=\{(t_1,\ldots,t_k):\ \mathbb{B} \vdash P(t_1,\ldots,t_k)\}$$

We say that a computation tree is tidy iff every call leaf l has its called node on the paths from the root to l.

Note that the computation tree on Figure 1 is tidy but that on Figure 2a is not.

A logic program $\mathbb{A}$ is tidy for a predicate P iff $\mathbb{A}$ has a tidy computation tree for P.

Let T be a computation tree of $\mathbb{A}$ for P. Suppose T is not tidy. We say that a called node is bad provided that one of its calls causes untideness of T /bad call/ - e.g. the node referred to by call /1/ in Figure 2a is bad. Namely, a node v of T is bad iff there exists a leaf referring to v which is on a different branch than v.

The untideness of T can be characterized by a pair $\langle \alpha_1, \alpha_2 \rangle$ of natural numbers, denoted $\alpha(T)$, such that α_1 is the maximal depth of all bad nodes of T, α_2 is the number of all bad nodes of T of the depth α_1.

This characterization allows to identify tidy trees, since T is tidy iff $\alpha(T) = \langle 0, 0 \rangle$. We shall use the lexicographic well-ordering $\prec$ of pairs of natural numbers.

Lemma A Let $\mathbb{A}$ be an untidy computation tree of a logic program $\mathbb{A}$ for P. Then there is a transformation of $\mathbb{A}$ to a program $\mathbb{A}'$ with a computation tree A' for P such that

/i/ $\mathbb{A}'$ extends $\mathbb{A}$

/ii/ $\alpha(A') \prec \alpha(A)$.

Corollary Given a predicate P, every logic program $\mathbb{A}$ can be transformed to a program $\mathbb{B}$, which extends $\mathbb{A}$ and is tidy for P.

Proof of the corollary: The transformation from Lemma A produces a program $\mathbb{A}'$ which extends

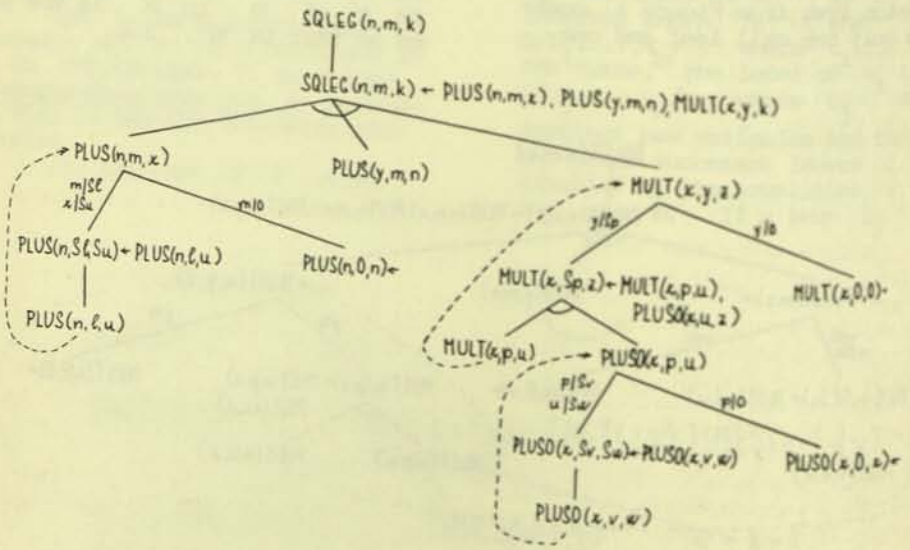


Figure 2b

$\mathcal{A}$ with a computation tree A', the α-characteristics of which is smaller than that of $\mathcal{A}$. Since $\prec$ is a well-ordering and the extension property of programs is transitive, it is clear that the iteration of this transformation gives a tidy program $\mathcal{B}$ extending $\mathcal{A}$ after a finitely many steps.

Sketch of the proof of Lemma A: Let $\alpha(A)=\langle\alpha_1,\alpha_2\rangle\neq\langle 0,0\rangle$. Let n be one of the bad nodes of A with the maximal depth α_1. Let $l_1,\ldots,l_k$ be all call leaves referring to n, the calls of which are bad. Denote the parent nodes of $l_1,\ldots,l_k$ by $m_1,\ldots,m_k$ respectively. Denote by B the subtree of A rooted in n /see Figure 3a/.

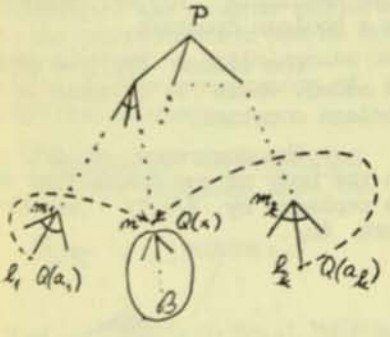


Figure 3a

Let us assume for simplicity that the root of B is labelled by a unary predicate Q. Let B^0 be obtained from B by renaming of variables in such a way that the label of the root of B^0 is $Q/z/$, where z does not occur in A. Let B^i be a tree obtained from B^0 by attaching an index i to all occurences of those predicates which differ from the labels of:

- all leaves of B referring to nodes outside of B,
- primitive leaves of B.

Let A' be obtained from A by

- attaching the tree B^i to every node l_i and cancelling the call from l_i to n
- replacing the occurence of $Q/a_i/$ in the label of m_i corresponding to the node l_i by $Q_i/a_i/$
- adding the successor $z=a_i$ to the AND-node m_i /this step can be avoided by proper renaming of variables in B^i whenever a_i is a variable/ for every $i\leq k$ /see Figure 3b/.

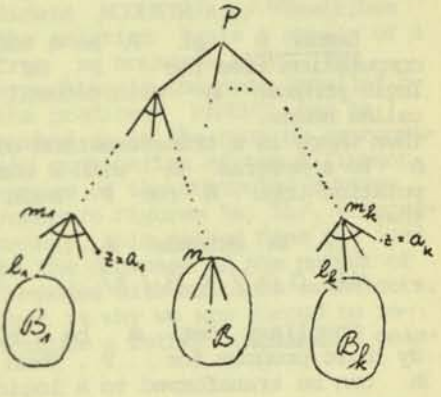


Figure 3b

This construction is illustrated on the program from Figure 2a, 2b. It is easy to see that A' has all the properties stated in lemma A.

3 PUSHING UP A CALLED NODE

The number of called nodes seems to be one of natural measures of the complexity of tidy logic programs. We shall show that this measure can be collapsed to 1. We shall use a method similar to that of Section 2.

Let n be the root of the

minimal subtree of T, which contains all called nodes of T and has an OR-node as a root. We call any OR-node between n and a called node of T <u>supercalled node</u>.

We characterize any tidy computation tree T by a triple

$$\beta/T/ = \langle \beta_0, \beta_1, \beta_2 \rangle$$

of natural numbers, where β_0 is the number of all supercalled nodes of T, β_1 is the maximal depth of all its called nodes and β_2 is the number of all its called nodes of the depth β_1. Obviously, T has a single called node iff $\beta_0 = 1$.

<u>Lemma B</u> Let A be a tidy computation tree for P of a logic program $\mathbb{A}$ with several called nodes.
Then there is a transformation of $\mathbb{A}$ to a program $\mathbb{A}'$ with a computation tree A' for P such that

/i/ $\mathbb{A}'$ extends $\mathbb{A}$,

/ii/ $\beta/A'/ < \beta/A/$.

<u>Corollary</u> Let $\mathbb{A}$ be a tidy logic program for P. Then $\mathbb{A}$ can be transformed to a logic program $\mathbb{B}$, which extends $\mathbb{A}$ and has a computation tree for P with a single called node.

This corollary follows from lemma B in the same way as the corollary of lemma A follows from lemma A.

<u>Sketch of the proof of lemma B</u>: Let

$$\beta/A/ = \langle \beta_0, \beta_1, \beta_2 \rangle$$

and $\beta_0 > 1$. Let n be one of the called nodes with depth β_1. Obviously, if $\beta_0 > 1$ then $\beta_1 > 1$, too. Let m be the first OR-node above n. Let n,m be labelled by $Q/x/$ and $R/y/$ respectively /see Figure 4a/.

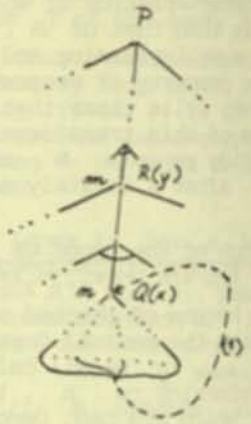


Figure 4a

Let $\mathbb{A}_1$ be a program obtained from $\mathbb{A}$ as follows

- the predicate $R/y/$ is replaced everywhere by a new binary predicate $\bar{R}/y,F/$, where F is a boolean constant.

- the clause $\bar{R}/y,T/ \leftarrow Q/y/$ is added, where T is again a boolean constant.

- the occurence of $Q/t/$ in the body of any clause from A is replaced by $\bar{R}/t,T/$ /see Figure 4b/.

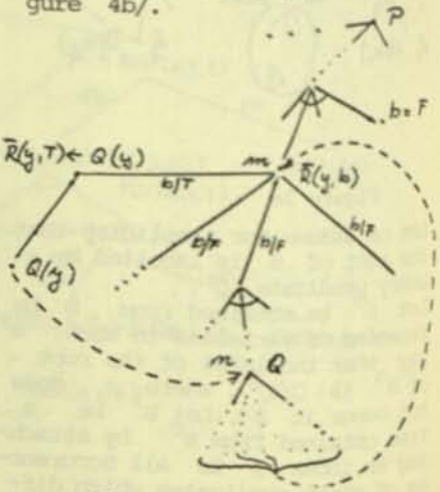


Figure 4b

It is not difficult to realize than $\mathcal{A}_1$ extends $\mathcal{A}$. Let $\mathcal{A}_2$ be a program which is a tidy extension of $\mathcal{A}_1$ obtained by removing the only bad call /1/ from the computation tree of $\mathcal{A}_1$ /see Figure 4b/ by the method of lemma A. Let A_2 be its computation tree.
Now the proof is complete provided that m is not the root of A. In the other case, the predicate P coincides with R. Then we have to add the clause

$$R/y/ \leftarrow \overline{R}/y,F/$$

to $\mathcal{A}_2$ to obtain $\mathcal{A}'$. It is obvious that $\mathcal{A}'$ extends $\mathcal{A}$. The tree A' is obtained again from A_2.
The proof of the fact that $\beta/A'/ \preceq \beta/A/$ is a mere technicality.

Remark The assumption about tidyness of the program subjected to the transformation can be dropped. But then no claim can be made on the β-characterization of the resulting program.

4 PUSHING UP OR-BRANCHING

We have just seen that the number of called nodes of a program does not reflect the complexity of the relations the program expresses. Our present interest will be in the minimization of the maximal number of alternations of AND- and OR-nodes on a branch of a computation tree. We shall prove that even this measure can be collapsed to 2.
First, we shall prove that branching in an OR-node, which is not a called node, can be pushed closer to the root. Then we notice that non-called OR-nodes with a single successor can be avoided.

The idea is illustrated by the self-explanatory example /see Figures 5a, 5b and 5c/. The predicate MCOUSIN/x,y/ describes the relation "y is a cousin of x from x mother's side". The branching in the node labelled by the predicate PARENT can be pushed up to the root by appropriate combination of two different copies of the contoured subtree /compare Figures 5a, 5b/. Unfortunately, this method does not lead to the decrease of the number of OR-nodes with multiple successors. That is why we are forced to introduce a rather complicated meas-

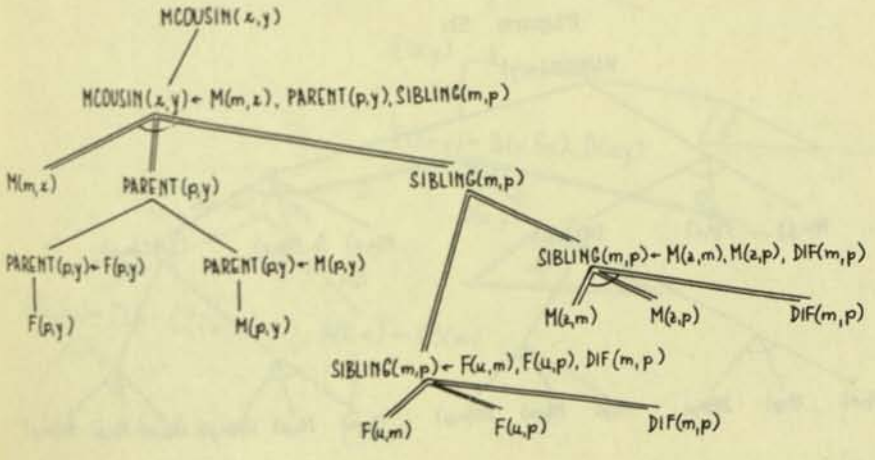


Figure 5a

sure φ on the nodes of the computation trees. Its purpose is to characterize the complexity of the path from the root to the given node in terms of intervening OR-nodes with several successors.

Let T be a computation tree, the single called node of which is the root. We say that there is multiple branching below an OR-node v of T iff there is an OR-node with multiple branching in the subtree of T rooted in v. It allows us to define valuation ν on every edge e of T. We proceed as follows

- if the upper node of e is an OR-node, we set $\nu/e/ = 1$ whenever this node has several successors, we set $\nu/e/ = 0$ otherwise

- if the lower node of e is an OR-node and if there is no multiple branching below this node then $\nu/e/ = 0$, otherwise $\nu/e/$ is the number of all those immediate successors of the upper node of e bellow which there is a multiple branching.

The weight $\varphi/v/$ of a node v of T is the sum of the weights of all the edges on the path from the root to v.

The branching of a computation tree T, where the root is the only called node, can be characterized by the pair $\varphi/T/$

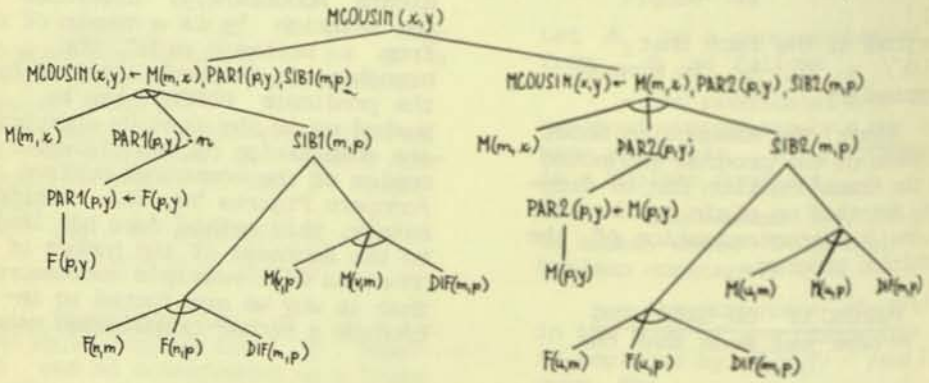


Figure 5b

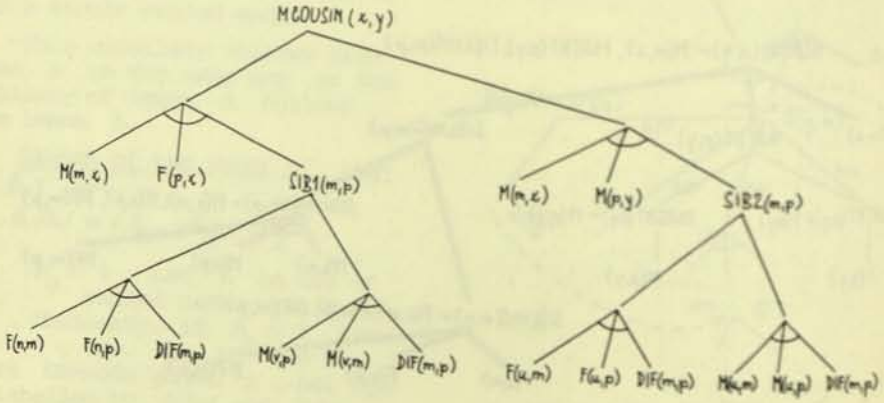


Figure 5c

of natural numbers γ_0 , γ_1 where γ_0 is the maximal weight $\varphi(v)$ of a node v of T and γ_1 is the number of all nodes n such that $\varphi(n) = \gamma_0$ and $\varphi(m) < \varphi(n)$ for every node m above n.

Lemma C Let A be a computation tree for P of a program $\mathbb{A}$, such that its root is the only called node. Then A can be transformed to a program $\mathbb{A}'$ with a computation tree A' for P such that

(i) $\mathbb{A}'$ extends $\mathbb{A}$

(ii) $\gamma(A') \leq \gamma(A)$

Corollary Let $\mathbb{A}$ satisfy the assumptions of Lemma C. Then $\mathbb{A}$ can be transformed to a program B extending $\mathbb{A}$ with a computation tree B for P such that no OR-node different from the root is called as well as no such node has more than one successor.

Sketch of the construction for lemma C : Let m_0 be such a node of A that one of its sons n has the maximal weight and $\varphi(n) = \gamma_0 = \varphi(A) > \varphi(m_0)$. In such a case m_0 must be an OR-node with several successors. Generally m_0 may have a sibling m_1 with several successors, too. (see Figure 6a). The transition of Figure 6a to 6b demonstrates the basic steps of the process of pushing up the branching of m_0 to the node k labelled by E . We proceed as follows

1. we tear off the subtree starting in the edge l_1

2. we make a new copy of the contoured subtree and we attach to it appropriately the subtree cut off at the step 1.

3. the subtree from the step 2. is attached to the node k .

Let A_1 be the resulting computation tree (Figure 6b). It is obvious that $\mathbb{A}_1$ extends $\mathbb{A}$. The weights of those edges which are changing during the process are indicated on Figures 6a, 6b. Obviously $\gamma(A_1) \leq \gamma(A)$. The more complex cases are treated similarly.

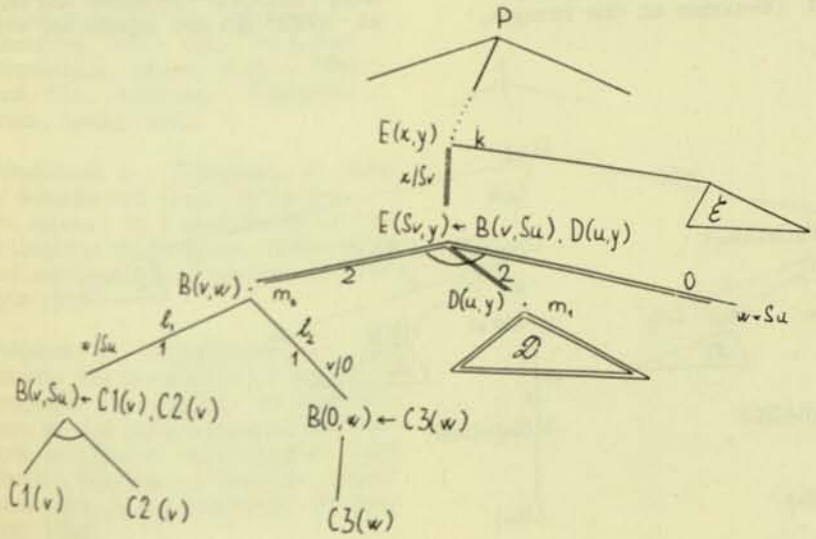


Figure 6a

Lemma D Let $\mathbb{A}$ be a logic program such that its computation tree A for P has a single called node and a single OR-node with several successors - both are the root of A.

Then there is a program $\mathbb{B}$ such that

- the maximal depth of the computation tree for $\mathbb{B}$ and P is two
- P has the same denotation in $\mathbb{B}$ and $\mathbb{A}$.

Proof : The non-called nodes without multiple branching can be avoided similarly as the node n in Figure 5b /see Figure 5c/.

4 NORMAL FORM THEOREM AND ITS APPLICATIONS

By the combination of the above methods, we can prove

Theorem Let $\mathbb{A}$ be a logic program computing the relation P. Then there is a program $\mathbb{B}$ computing the same relation P with a tidy computation tree B, which has at most one called node and at most two alternations of AND- and OR-nodes on one branch.

The extensive use of computation trees in Sections 2 - 4 demonstrates that graphical description of logic programs provides deep insight into their structure. For example, the binary programs /see Tärnlund 1977/, stratifiable programs /see Šebelík and Štěpánek 1982/ or recursion-free programs have certain characteristic types of computation trees. Many structural properties of logic programs are easily recognizable in computation trees, which help to detect those parts of programs calling for special attention or optimization. Computation trees clearly visualize the dependencies between predicates of a given program and thus make it possible to recognize those subgoals which can be solved concurrently.

We have suggested several methods how to modify computation trees to obtain better organized programs computing the same relation. In a subsequent paper /Štěpánek and Štěpánková/, we will use them to prove that the syntactical restrictions of the language PRIMLOG /Markusz and Kaposi 1982/ do not impose any sig-

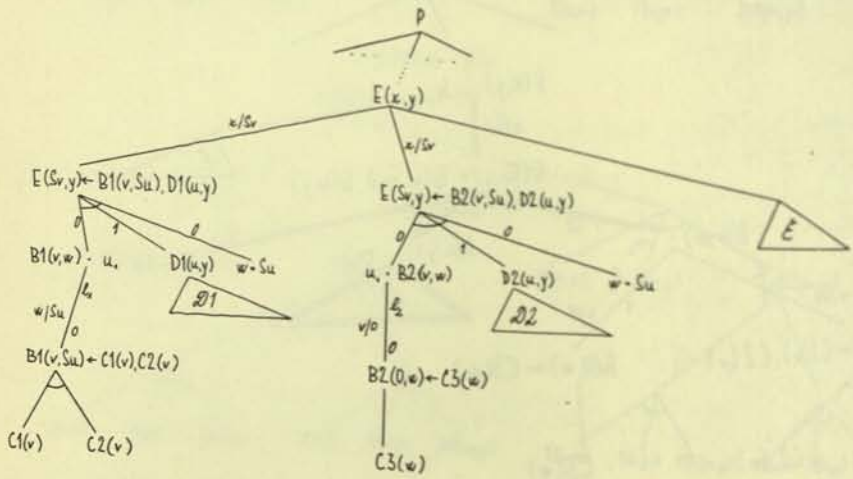


Figure 6b

nificant restrictions on the class of computable functions.

REFERENCES

Greibach, S.A. Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science, Vol. 36, Springer-Verlag, Heidelberg, 1974

Harel, D. And/Or Programs: A New Approach to Structured Programming ACM Transactions on Programming Languages and Systems 2 , No. 1, 1 - 17 , 1980.

Harel, D. On And/Or Schemes. in Mathematical Foundations of Computer Science 1980, P. Dembinski /Editor/, Lecture Notes in Computer Science, Vol. 88 , Springer-Verlag, Berlin, 246 - 260, 1980.

Markusz, Z., Kaposi, A.A. A Design Methodology in Prolog Programming, in Proc. 1st International Logic Programming Conference, Marseille. Van Caneghem /Editor/, 139 - 145, 1982.

Šebelík, J., Štěpánek, P. Horn Clause Programs for Recursive Functions, 325 - 340, in Logic Programming, Clark, K.J., Tärnlund, S.A. /Editors/, Academic Press, London 1982

Štěpánková, O., Štěpánek, P. And/Or Schemes and Logic Programs. /to appear/ in Proceedings of the Colloquium on Algebra, Combinatorics and Logic in Computer Science, Györ 1982

Štěpánek, P., Štěpánková, O. Improving the Structure of Logic Programs. /to appear/ in Proceedings of 3rd International Conference Artificial Intelligence and Control Systems of Robots, Smolenice 1984, North Holland, Amsterdam 1984.

Tärnlund, S.A. Horn Clause Computability, BIT 17, No. 2, 215 - 226, 1977.

Van Caneghem /Editor/ Proceedings of the First International Logic Programming Conference, Marseille 1982. Groupe Intelligence Artificielle et Association pour la Diffusion el le Dévelopment de PROLOG 1982.

Van Emden, M.H., Kowalski, R.A. The Semantics of Predicate Logic as a Programming Language. J ACM 23, No. 4, 733 - 742, 1976.

SEMANTIC INTERPRETATION FOR THE EPISTLE SYSTEM

Michael C. McCord
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

ABSTRACT

EPISTLE is a natural language processing system being developed at IBM Research, with current application to text-critiquing: criticism of grammar and style in documents. The EPISTLE grammar, with a very broad coverage, can be considered purely syntactic. This paper describes a semantic interpretation component, SEM, written in PROLOG, which will be useful in further developments for the system. SEM is based partly on previous work by the author, but the present system is different in that it translates surface parses to logical forms in a single stage, in which there is interleaving of the processes of sense selection, slot filling, other types of modification, movement of nodes, and exercising of semantic constraints. Furthermore, the constraints used are not simple type-checks, but involve inference with world knowledge.

1 INTRODUCTION

EPISTLE (Miller, Heidorn and Jensen, 1981, Heidorn et al., 1982) is a natural language processing system applied currently to text critiquing: Authors preparing a document will be able to use EPISTLE to get corrections and criticism of grammar, spelling, and style in the text. Other applications to document analysis and generation are planned for the system. EPISTLE uses a grammar (Jensen and Heidorn, 1983) written in the NLP rule language (Heidorn, 1972) and a lexical/morphological component (Byrd, 1983) which together give the system a very broad coverage of English. The grammar can be considered to be purely syntactic, using no semantic constraints and producing purely syntactic analyses of sentences.

This paper describes a semantic interpretation component, SEM, written in PROLOG, which takes the output of the syntactic component and produces logical forms for sentences. SEM will be useful for refinements of the text-critiquing application, and will be crucial for certain planned applications, such as document indexing and expert systems associated with text analysis.

SEM has some elements in common with previous semantic interpretation systems of the author (McCord, 1982, 1981). The logical language used as the target of interpretation is much like that in (McCord 1981), including focalizers. Scoping problems are dealt with. However, the present system is different in that it is organized into a single stage in which there is interleaving of the processes of sense selection, slot filling and other types of modification, movement of nodes, and exercising of semantic constraints. In many natural language systems, the semantic constraints used are simple type-checks in a hierarchy of types. It is argued in this paper that this is not adequate

generally, and that general inference with world knowledge is needed during semantic interpretation. Such a mechanism is used in SEM.

2 NATURE OF THE INPUT TO SEM

In the interests of modularity and broad coverage, the approach of the EPISTLE grammar is to be as independent of semantics as possible, and to produce syntactic analyses which in themselves often have enough information for useful text critiquing. When semantics (and pragmatics) are ignored in a natural language system, sentences can be extremely ambiguous. For avoiding multiple analyses, the design of the EPISTLE grammar includes the idea of the **approximate surface parse.** For most sentences, a single, "approximate" syntactic analysis is produced. Achieving this involves, mainly, two decisions.

One decision is that modifiers are attached in canonical ways. For example, postmodifying prepositional phrases are normally attached to the next higher node which is a verb phrase or clause. With this decision, the following example has only one analysis.

John saw the man in the park with the telescope.

Another decision is to ignore, in most cases, the identification of "deleted" or "moved" items. Thus, in

Which horse did you want to win?

no indication is made of the subject or object of "win", and, in fact, no "trace" is shown at all for "which horse". Consequently, this sentence is given only one analysis. Similarly, in

John was killed by the river.

there is only one analysis, with no indication of the logical subject of "killed" ("by the river" is simply a prepositional phrase postmodifying "killed").

Interestingly, the idea of the approximate surface parse is rather similar to F. Pereira's **right-most normal form** (Pereira 1983), which was designed for the same purpose (reducing ambiguities in syntax). These ideas were arrived at independently. Pereira's analyses do contain more information pertinent for semantics, for instance the indication of traces, produced by use of an extraposition grammar (Pereira 1981).

For SEM, an interface from NLP to PROLOG produces, for each syntactic analysis, a PROLOG term of the form

syn(Features,Marker, Head,Daughters)

which we call a **syntactic item.** Here **Features** is a list of terms representing the syntactic features of the sentence (or phrase) being analyzed. **Marker** is a variable which relates the item to copies of it made by SEM, through unification with the markers of the copies (SEM can make copies for handling deleted and moved phrases). In the input to SEM, no two markers are unified. **Head** is (the root form of) the head word of the phrase. (The grammar has the flavor of a dependency grammar and every phrase has a head word.) Morphological features of **Head** are included in **Features. Daughters** is a list of **syns** representing the modifiers of the head word (its daughters in the analysis tree). The position of the head is indicated in this list.

3 OUTPUT OF SEM

SEM takes syntactic items and produces logical forms representing

the meanings of sentences. These forms are built up from variables, constants, and compound terms consisting of a predicate (usually a sense of some word appearing in the sentence) with its arguments, or a conjunction of forms. Some predicates can have logical forms as arguments. This is the case for (senses of) verbs like "believe". Quantifiers, like "each" and "many", and other focalizers (cf. McCord 1981), like "only" and "even", are also considered higher-order predicates in the system which happen to take (two) logical forms as arguments.

As an example, the logical form produced by SEM for the sentence

Who does Mary believe that every man likes?

is

```
wh(X,person(X)&
   believe(E1,mary,
    every(man(Y),like(E2,Y,X))))
```

The first argument of every noun or verb sense, such as X for "person" and E1 for "believe", is called the **entity argument**, and stands for the event, state of affairs, or individual referred to by the predication. Any free variables (such as E1 and E2) in a logical form are considered to be existentially quantified.

Other examples of logical forms will be given below. For more discussion of the logical language being used here, see (McCord 1981).

4 SEMANTIC ITEMS

In barest outline, the main procedure of SEM converts each node of a **syn** tree to a logical form (representing a sense of the head of the node), and combines these forms to make a logical form for the whole sentence.

However, in doing the combining, richer structures, called **semantic items**, are actually manipulated. A semantic item is of the form

sem(Features,Connector, Marker,LogicalForm).

Here, **Features** and **Marker** are as in syntactic items, with the additional condition that for noun phrases and clauses, the **Marker** is unified with the entity variable for the head predication in the **LogicalForm**. In the initial semantic item created for a node, the **LogicalForm** is normally a simple predication (corresponding to a sense of the head word); but, after modification by (combination with) other semantic items, this field becomes ever more complex.

The **Connector** is a term which, roughly, determines how the semantic item can combine with other semantic items in the process of modification. The procedure **mod**, described below, which allows one semantic item to modify (or combine with) another to produce a third, keys mainly off the connectors of the first two items. Typically, a connector term contains variables which are (unified with) argument variables of the head predication in the semantic item; and the structure of the connector term (as interpreted by **mod**) determines how these arguments get filled. A special case of a connector is a slot frame, and slot filling for verbs and nouns is handled in SEM by **mod**.

Examples of semantic items are

```
sem(quant:..,Q/P,nil,each(P,Q)).

sem(quant:..,%Q,nil,
     each(man(X),Q)).

sem(vp:..,nil-(subj:X)-(obj:Y),
     E,see(E,X,Y)).
```

```
sem(vp:..,nil-(subj:Y)-(pobj(by):X),
     E,see(E,X,Y)).
```

In the first item, the connector Q/P is such that modification by the item results in (1) unifying P with the logical form of the modificand, and (2) creating a new item like the second item above. This second item has a new connector %Q which can "cause" unification of Q with a further modified logical form. The third and fourth items have connectors which are slot frames. (The format for these is slightly simplified.) In the fourth one, the slot frame has undergone a transformation which would be appropriate for a passive VP.

In (McCord 1981), semantic items were terms with slightly less information, containing only the **Connector** and **LogicalForm** fields. Connectors were called **operators**. The new name is more appropriate, especially in the new system, where connectors can be slot frames, because **mod** can use the connectors of both a modifier and its modificand: The control is more symmetric.

5 OVERVIEW OF THE INTERPRETATION PROCEDURE

There are four main ingredients in the interpretation procedure: **sense selection, modification, transformations,** and **knowledge-checking.** The main procedure (called **semant**), acts recursively on the nodes of a **syn** tree, and uses all four of these ingredients at every level.

Sense selection is done by calling a procedure **sense**, which takes the features, marker, and head of the **syn** node, and returns an initial semantic item for the node. The choice made by **sense** is nondeterministic. Wrong choices may get filtered out by the other three ingredients named above. Sense selection is discussed further in Section 6.

Modification, residing in the procedure **mod**, is the heart of the interpretation process. As indicated in the preceding section, **mod** allows one semantic item to modify (or combine with) a second to produce a third. As for which pairs of items are combined by modification, the basic, simplified idea is that all the daughters of a node modify the node (with the leftmost acting as outermost modifier), after the daughters themselves have been interpreted and a sense for the given node is chosen. Modification is discussed further in Section 7.

Transformations are needed in this scheme because the structure of the syntactic analysis tree may need "correcting" in order to make the straightforward process of modification work correctly. There are two sources of this need for correction.

One source is that quantifiers (and several other types of modifiers) may have intended scopes in logical form which do not correspond to their positions in the syntactic structure. This problem was discussed extensively in (McCord, 1982, 1981) and was dealt with there by a type of tree transformation called **reshaping**. In these previous systems, reshaping was done in a whole separate stage (on the whole tree), before any modification was done. In the present system, the steps of reshaping are interleaved with all the other steps of interpretation, so that there is only one stage of interpretation. This is done so that there can be more "immediate feedback" and filtering for the choices of reshaping from the other

steps of interpretation, especially knowledge-checking.

The other source of the need for correction is the fact that the syntactic analysis tree is an approximate surface parse, so that modifiers may need to be moved, created, or identified in some way.

Transformations appear in several different ways in SEM. There are some (though very few) like ordinary transformations of transformational grammar, operating on whole **syn** trees. The others are more implicit. In the (reshaping) transformations dealing with scoping, semantic items corresponding to original tree nodes are moved, but their positions are kept track of in arguments of PROLOG procedures. Still another type of transformation is of slot frames, handled by the procedure **sense**. These various types will be discussed below where they are pertinent.

Knowledge-checking is a generalization of semantic type-checking, in which the reasonableness of a logical form is checked, with inference, against knowledge about the use of the predicates in the form. At every level of call to **semant**, the logical form of the semantic item produced at that level is checked with the procedure **kcheck**, which is discussed in Section 8.

Now let us look at the definition of the main procedure **semant**.

```
semant(Syn,Sem,Sisters) <-
  transform(Syn,Syn1) &
  semant(Syn1,Sem,Sisters).

semant(syn(Features,E,Head,Daus),
       Sem,Sisters) <-
  semantlist(Daus,Mods) &
  reorder(Mods,Mods1) &
  sense(Features,E,Head,Sem0) &
  modlist(Mods1,Sem0,
          Sem,Sisters) &
  satisfied(Sem) &
  kcheck(Sem).
```

The top-level use of **semant** is to take a syntactic item Syn and produce a semantic item Sem. However there is an additional output, Sisters, which is important for lower-level calls. Interpretation of a node Syn can produce new (left) sisters for it because of the operation of **raising**. This is a type of transformation involved in reshaping (to handle scoping problems). For example, the quantifier node "each" in the noun phrase "each man" is raised to become a sister of the noun phrase. Raising is handled by **mod** and will be discussed further in Section 7. In the top-level call to **semant**, the Sisters list is required to be nil.

The first clause defining **semant** calls the procedure **transform** to perform a tree transformation on Syn, and then calls **semant** again on the output. (Thus another transformation could apply, and so on.)

Some of these transformations are like the transformations of transformational grammar, although SEM needs only a very few. The only transformations of this type in the current version of SEM are wh-movement for wh-questions and relative clauses. However, coordination (with ellipsis) will probably be treated in SEM by use of transformations. (This will therefore be an alternative to the metagrammatical, parsing approach to coordination in (Dahl and McCord, to appear).)

Other transformations performed by **transform** have the purpose of trying out corrections to the approximate surface parse. For

example, postmodifying prepositional phrases can be reattached. The non-determinism of PROLOG allows wrong reattachments to be blocked by other ingredients of interpretation.

The various transformations of these two sorts are defined simply by PROLOG clauses for **transform**, one clause per transformation.

Now let us look at the second clause for **semant**. The call to **semantlist** does the recursive interpretation on the daughters (Daus) of the syntactic item, producing a list Mods of semantic items which are to be the (semantic) modifiers of the node. This procedure just calls **semant** itself on each member of the list Daus, and any sisters produced are blended into the list Mods during the process.

The procedure **reorder** is a reshaping procedure (dealing with scoping). It performs a permutation of the Mods list according to scoping heuristics (see McCord 1982, 1981). Again, wrong choices could get filtered out by other ingredients.

As indicated above, **sense** selects a semantic item (Sem0) representing a sense of the node being worked on.

Now the sense Sem0 for the node can be modified by its modifiers Mods1. This is done by the call to **modlist**, which calls **mod** for each member of Mods1, so that these items act on Sem0. The leftmost is the outermost modifier; this means procedurally that the rightmost actually modifies Sem0 first, the next-to-rightmost modifies the result, and so on.

The last two procedure calls, to **satisfied** and to **kcheck**, act as filters on the result Sem of the interpretation. The first one blocks Sem if the connector of Sem is a slot frame containing (unfilled) obligatory slots. The procedure **kcheck** is discussed in Section 8.

6 SENSE SELECTION

In the call

sense(Features,E,Head,Sem),

the list Features, the marker E, and the head word Head are used as input, and the output is the semantic item Sem representing a sense of Head as head word of a phrase of type Features with marker E. Another way of saying it is that Sem is an initial semantic item for the given phrase-with-head before anything has modified it. The logical form of Sem is a predication whose predicate is one of the senses of Head (for the given Features); and the connector, depending also on Features, controls how the arguments of this predication get filled in. For a given word sense, different connectors can be produced by **sense**, because the connector can depend on non-lexical features of the phrase. For example, a passive VP **syn** gets a different **sem** from an active one.

Currently, **sense** produces ten different types of connectors. Examples of three different types were given in Section 4. These will not be described systematically in this paper, because most of them are like connectors already described in (McCord 1981). However, slot frames were not used as connectors in (McCord 1981), and these are worth describing here, especially because **sense** has to do a bit of work to produce them.

A slot frame is of the form

frame(Type,Slots).

Here, **Slots** is a list of slots, each of which is a pair (as in (McCord 1982)) **Slotname:Marker.** Because of the way the procedure **mod** is applied, slots get filled right-to-left. So for convenience in displaying slot lists, these lists are formed with the left-associative operator '-'. Thus, examples of slot lists are

nil-(subj:X)-(obj:Y)-(pobj(to):Z)

nil-(subj:X)-(iobj:Z)-(obj:Y)

Each of these could get associated with a predication like give(E,X,Y,Z) for a ditransitive verb.

The **Type** field for a slot frame is either **nil** or is of the form **adjunct(X).** The latter type is used when the semantic item comes from a phrase like a prepositional phrase, a participial clause, or a relative clause, where there is a "topic" X (the first argument of the preposition, the missing subject in the participial clause, the topic of the relative clause) which will be unified with the marker of the modificand when this semantic item acts as an adjunct modifier.

For phrases whose head is a verb, **sense** does the following things to find a corresponding **sem.**

First a transitivity type (transitive, ditransitive, etc.) is obtained from the features. (This is actually non-deterministic, because some verbs can have more than one transitivity feature.) The voice (active, passive) is also determined from the features. For the given transitivity type and verb, a predication Pred corresponding to a sense of the verb, and a canonical slot list Slots, are made up (assuming an active clause environment for the verb).

Then a procedure **slotrans** looks at the canonical slot list Slots and the voice, and produces a transformed slot list Slots1 (which could be left the same as Slots itself). For example, in the passive case, the slot list

nil-(subj:X)-(obj:Y)

is transformed to

nil-(subj:Y)-(pobj(by):X).

This operation is non-deterministic; e. g., for ditransitive verbs more than one result is possible.

Finally, a procedure **mkframe** looks at the phrase category and Slots1, and makes the frame that will be the connector for the desired **sem.** In the case of participial clauses, **mkframe** deletes the subject slot from Slots1, so that in this clause no overt subject is sought. The marker X associated with this deleted subject slot is, however, stored in the type, adjunct(X), of the frame. In the case of imperative clauses, **mkframe** also deletes the subject slot, but unifies its marker variable with "you".

As an example, for the participial clause "given me by my aunt", the successful choices would be the following. The predication is give(E,X,Y,Z), and the canonical slot list is

nil-(subj:X)-(obj:Y)-(pobj(to):Z).

This is transformed by **slotrans** to

nil-(subj:Y)-(iobj:Z)-(pobj(by):X).

Then **mkframe** produces the frame

frame(adjunct(Y),

```
      nil-(iobj:Z)-(pobj(by):X)),
```

and the **sem** produced by **sense** is

```
sem(ptprtcl:..,
  frame(adjunct(Y),
    nil-(iobj:Z)-(pobj(by):X)),
    E, give(E,X,Y,Z) ).
```

7 MODIFICATION

The procedure **mod** handles the interaction between a semantic item Sem and a semantic item Sem0 which has been made (implicitly) a daughter of Sem by **semant**, so that Sem0 is a candidate to modify Sem. There are actually three types of interactions:

(1) The daughter Sem0 can simply modify the mother Sem, producing a new (modified) version Sem1 of the mother. In this case, the daughter "goes away" (no longer is used in the interpretation). This is the case, for example, when the daughter fills a slot in the mother, or is an adjunct modifier such as a relative clause.

(2) The daughter Sem0 is simply raised to become a (left) sister of the mother Sem, and no real modification takes place. This happens, for example, within the processing of "the grades of each student", where the quantifier "each" (after already having modified "student") gets raised to a left sister of "the grades" (so that it has wider scope).

(3) Both modification and raising take place. Both the daughter and the mother can get changed, and the new version of the daughter gets promoted to be a left sister of the mother. This happens when the quantifier "each" modifies "student" in the noun phrase "each student". The raised version of the daughter has a logical form each(student(X),Q) and a connector %Q so that it is ready to modify its new mother (by unifying Q), or to be raised even further, as in (2). The "student" node still needs to be there for slot filling. It has a slightly different connector, so that it does only slot filling for its mother. (If there had been no determiner, the original noun phrase would modify its mother both by slot filling and by left-conjoining.)

The procedure **mod** manages these three types of interactions by having the calling form:

```
mod(Sem0,Sem,Sem1,Sisters,Sisters1).
```

Here Sem0 is the daughter, Sem is its mother, and Sem1 is the new version of the mother. The last two arguments are treated as a difference list (for convenience of the calling procedure **modlist**) which contains the raised daughter in cases (2) and (3) above and is empty in case (1). Case (2) is handled by calling a procedure **above** which is like the procedure of that name in (McCord 1981). Everything else is handled by a series of clauses for **mod** which look mainly at the connectors of Sem0 and Sem (especially Sem0). These are like clauses for **trans** in (McCord 1982, 1981), except that they now handle slot filling and raising, as well.

A sample clause of this type, which illustrates case (3) above, is

```
mod(sem(Feas0,P/Q,X0,LF0),
    sem(Feas,*,X,Q),
    sem(Feas,nil,X,Q),
    S, sem(Feas0,%P,X0,LF0):S ).
```

Here, the logical form LF0 in the daughter could be each(Q,P), and the logical form in the mother could be student(X), so that Q gets unified with student(X).

One of the clauses for **mod** calls a procedure **filler**, which handles slot filling. This procedure expects the connector of the mother node to be a slot frame. From the slot list, a slot is chosen, looking from the right (since modifiers do their work right-to-left). This choice is non-deterministic. Slots can be passed over, but only if they are not declared to be obligatory. Any slots passed over, plus the chosen slot itself, are discarded from the slot list (in the sense that the new version of the mother has a slot frame with these slots removed). The marker of the chosen slot is unified with the marker of the modifier. (This is the main point of slot filling.) Then, to check on the correctness of this filling, **filler** calls a procedure **fill**, which looks at the name of the chosen slot and knows what specific slots require of their fillers. For example, the slot pobj(by) would require a prepositional phrase whose preposition is "by".

The action of **fill** can also unify other marker variables. For example, in the clause

John was asked to see Bill.

the main verb gets a predication ask(E,X,Y,Z) (read "X asks Y to Z"), and the associated slot list is

```
nil-(subj:Y)-(pobj(by):X)-
    (infcomp(Y):Z).
```

At the time of filling the infcomp(Y) slot, the infinitive complement "to see Bill" has been interpreted and has semantic item of the form:

```
sem(infcl:...,
    frame(adjunct(U),nil), E1,
    see(E1,U,bill)).
```

The procedure **fill** knows to unify U with the variable Y in the slot infcomp(Y). After the subj:Y slot for "ask" is filled by "john", the resulting logical form for the whole sentence (neglecting tense) is

```
ask(E,X,john,see(E1,john,bill)).
```

In (McCord 1982, 1981), slot filling was done during parsing. In some ways, that is more natural. Indeed, it would be attractive to have only one pass (parsing) in which complete logical forms are produced, as has been done in the earliest logic grammars (see Dahl 1981). But there are strong arguments for having a second pass in which one can look at the whole parse tree. A good treatment of scoping is easier on a second pass, and coordination is probably easier to treat on a second pass (with transformations which duplicate elided material). The first-pass treatments of coordination in (Dahl and McCord, to appear) and (Woods 1973) involve looking at parse histories, which can be difficult to manage. A second-pass treatment of coordination would go hand-in-hand with slot filling on the second pass. The system of (Pereira 1983) is akin to SEM in that slot filling is done there on a second pass. However, in (Pereira 1983) there is a third pass, where scoping is treated and the final logical form is produced. It seems advantageous to have only one pass after parsing, in which all the steps are interleaved, so that logical forms get built up in the cycle and can serve as input at every level to checks like knowledge-checking. (An early version of SEM actually had three passes: slot filling, reshaping, and modification. No systematic comparisons of efficiency have been made, but it seems likely that immediate feedback from constraints

will produce greater efficiency. In addition, the current version is simpler is design.)

8 KNOWLEDGE CHECKING

In producing semantic interpretations, many choices can be made (selection of word senses, placement and action of modifiers, etc.). Some sort of guidance or filtering is needed. In many natural language systems, **semantic type-checking** is used for filtering: Senses of words (especially verbs) have a semantic type associated with each slot in their slot frames. Thus a sense seel(E,X,Y) of "see" might have the slot list

```
nil-(subj:X:animal)-(obj:Y:physobj).
```

Slot fillers are required to have types which match the slot type, perhaps after moving about in a hierarchy of types.

In logic grammars, the matching of types within a hierarchy can be implemented in a particularly powerful way by using unification of logic terms representing partially specified types (see Dahl 1981). (Types can be represented as lists like t1:t2:t3:*, where t1 is a supertype of t2, t2 is a supertype of t3, and the tail of the list is a variable.) An advantage of this unification approach is that type-matching requirements can be exercised in a top-down way during parsing (by PROLOG, for a definite clause grammar (Pereira and Warren 1980)).

Semantic type-checking may be adequate in small domains, but it appears not to be adequate in general. Consider sentences of the sort discussed by Bar-Hillel (1964):

The pen is in the box.

The box is in the pen.

Let us just consider two senses of "pen": "writing_pen" and "animal_pen". Most people would get the "writing_pen" interpretation in the first sentence and the "animal_pen" interpretation in the second. The disambiguation can be made by requirements of the "contained_in" sense of "in", together with knowledge about (normal) sizes of writing pens, animal pens, and boxes. For simplicity, we could say that the precondition for

```
contained_in(X,Y)
```

is

```
smaller(X,Y).
```

A frame with semantic types like

```
X:small, Y:large
```

will not do, because the requirement on X and Y is relative. Of course, in a limited domain, with a small number of types of objects, one could suitably enumerate the required pairs. But, in general, this cannot be done, and we must make a computation (an inference) not based simply on finite look-up. In fact, the two noun phrases in a sentence of the form

The ... is in the ... .

as in

The object weighing 800 pounds is in the pen.

could be very complex, and all of the information could be used to determine smaller(X,Y).

So, instead of doing simple type-checking, we need to do **knowledge-checking**, where logical forms are checked for reasonableness by doing more general inference with real-world knowledge. At

present in SEM, knowledge-checking is implemented in the following (approximate) way.

Recall that **semant** makes the call

```
kcheck(Sem)
```

at every level. There is a (partial) logical form LF in Sem, which is to be checked. As an example, if we are looking at the top level of

Each box is in a pen.

the logical form LF (with the "animal_pen" sense of "pen") is essentially

```
each(box(X),animal_pen(Y)&
     contained_in(X,Y)).
```

The next step (to make the knowledge-check easier to handle) is to strip LF of quantification, forming a conjunction of the remaining bases, as well as to remove any qualification in these bases. For the example, the stripped form is

```
box(X)&animal_pen(Y)&
      contained_in(X,Y)
```

where, as usual, the free variables are considered to be existentially quantified. Finally, we replace the head predication by its **precondition** (given by a unit clause for the predicate **precond**). This results in the form

```
box(X)&animal_pen(Y)&
      smaller(X,Y).
```

Finally, we pass this form to PROLOG to try to prove it, i. e., to find a case of a box X and an animal pen Y where X is smaller than Y. With this method, SEM does succeed in getting the reasonable disambiguations of "The pen is in the box" and "The box is in the pen".

This method of doing **kcheck** can be seen to be a true generalization of type-checking in a hierarchy of types. To illustrate the mapping, if the predication seel(E,X,Y) has the type requirements

X:animal, Y:physobj,

then we can give it the precondition

```
animal(X)&physobj(Y).
```

Given the sentence "John saw a star", **kcheck** will try to prove

```
star(Y)&animal(john)&physobj(Y).
```

Given clauses

```
man(john).
star(s1).
physobj(X) <- star(X).
animal(X) <- human(X).
human(X) <- man(X).
```

this proof will succeed. Conditional clauses of the sort given correspond to type-hierarchy relationships.

But this method of defining **kcheck** is probably only an approximation to what is needed. Working with the stripped logical forms is not sufficient. And perhaps instead of looking merely at preconditions, one should in general be testing for **consistency** of the logical form with the current knowledge base. There may be no clear distinction one could make between the requirements of preconditions and the general requirements of consistency, although one must be concerned with efficiency. (For a discussion of the role of consistency in information systems, see Kowalski 1979). This point of view for SEM will be investigated.

REFERENCES

Bar-Hillel, Y. Language and Information. Addison-Wesley, 1964.

Byrd, R.J. Word formation in natural language processing systems. Proc. 8th IJCAI, 704-706, 1983.

Dahl, V. Translating Spanish into logic through logic. AJCL, 13, 149-164, 1981.

Dahl, V. and McCord, M.C. Treating coordination in logic grammars. AJCL, to appear.

Heidorn, G.E. Natural Language Inputs to a Simulation Programming System. Naval Postgrad. School Tech. Report No. NPS-55HD72101A, 1972.

Heidorn, G.E., Jensen, K., Miller, L.A., Byrd, R.J. and Chodorow, M.S. The EPISTLE text-critiquing system. IBM Systems Journal, 21, 305-326, 1982.

Jensen, K. and Heidorn, G.E. The fitted parse: 100% parsing capability in a syntactic grammar of English. IBM Research Report RC 9729, 1983.

Kowalski, R.A. Logic for Problem Solving. North-Holland, 1979.

McCord, M.C. Using slots and modifiers in logic grammars for natural language. Artificial Intelligence, 18, 327-367, 1982.

McCord, M.C. Focalizers, the scoping problem, and semantic interpretation rules in logic grammars. Tech. Report, Univ. KY, 1981. To appear in Logic Programming and its Applications, D. Warren and M. van Caneghem, Eds.

Miller, L.A., Heidorn, G.E. and Jensen, K. Text-critiquing with the EPISTLE system: an author's aid to better syntax. AFIPS Conf. Proc., 50, 649-655, 1981.

Pereira, F. Extraposition grammars. AJCL, 7, 243-256, 1981.

Pereira, F. Logic for natural language analysis. SRI International, Tech. Note 275, 1983.

Pereira, F. and Warren, D.H. Definite clause grammars for language analysis - a survey of the formalism and a comparison with transition networks. Artificial Intelligence, 13, 231-278, 1980.

Woods, W.A. An experimental parsing system for transition network grammars. in Natural Language Processing, R. Rustin, Ed., 145-149, Algorithmics Press, 1973.

On Gapping Grammars

Veronica Dahl
&
Harvey Abramson

Department of Computer Science
Simon Fraser University
Burnaby, B.C. Canada

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

ABSTRACT

A Gapping Grammar (GG) has rewriting rules of the form:

$$\alpha_1, gap(z_1), \alpha_2, gap(z_2), \ldots,$$
$$\alpha_{n-1}, gap(z_{n-1}), \alpha_n \rightarrow \beta$$
$$\alpha_i \in V_N \bigcup V_T$$
$$G = \{ gap(z_1), gap(z_2), \ldots, gap(z_{n-1}) \}$$
$$z_i \in V_T^*$$
$$\beta \in V_N^* \bigcup V_T^* \bigcup G^*$$

where V_T and V_N are the terminal and non-terminal vocabularies of the Gapping Grammar. Intuitively, a GG rule allows one to deal with unspecified strings of terminal symbols called *gaps*, represented by $z_1, z_2, \ldots, z_{n-1}$, in a given context of specified terminals and non-terminals, represented by $\alpha_1, \alpha_2, \ldots, \alpha_n$, and then to distribute them in the right hand side β in any order. GG's are a generalization of Fernando Pereira's *Extraposition Grammars* where rules have the form (using our notation):

$$\alpha_1, gap(z_1), \alpha_2, gap(z_2), \ldots, gap(z_{n-1}), \alpha_n \rightarrow$$
$$\beta, gap(z_1), gap(z_2), \ldots, gap(z_{n-1})$$

i.e., gaps are rewritten in their sequential order in the rightmost positions of the rewriting rule. In this paper we motivate GG's by presenting grammatical examples where XGs are not adequate and we describe and discuss alternative implementations of GGs in logic.

1. Introduction

A grammar is a finite way of specifying a language which may consist of an infinite number of "sentences". A logic grammar has rules that can be represented as Horn clauses. Such logic grammars can conveniently be implemented by the logic programming language Prolog: grammar rules are translated into Prolog rules which can then be executed for either recognition of sentences of the language specified, or (with some care) for generating sentences of the language specified.

Since the development of the first logic grammar formalism by A. Colmerauer in 1975 (Colmerauer,1975), and of the first sizeable application of logic grammars by V. Dahl in 1977 (Dahl,1977), several variants of logic grammars have been proposed, sometimes motivated by ease of implementation (Definite Clause Grammars, DCGs, [Pereira&Warren,1980]), sometimes by a need for more general rules with more expressive power (Extraposition Grammars, XGs, [Pereira,1981]), sometimes with a view towards a general treatment of some language processing problem such as coordination (Modifier Structure Grammars, MSGs, [Dahl&McCord,to appear]), or of automating some part of the grammar writing process, such as the automatic construction of parse trees and internal representations (MSGs, op.cit; Definite Clause Translation Grammars, DCTGs, [Abramson,1984]). Generality and expressive power seem to have been the main concerns underlying all these efforts.

In this paper we present another logic grammar formalism called Gapping Grammars, GGs, which we believe to be the most general to date. We examine three possible

implementations, and discuss the adequacy of GGs for certain language processing problems that cannot be expressed as easily in any other formalism.

GG rules can be considered as meta-rules which represent a set (possibly infinite) of ordinary grammar rules. They permit one to indicate where intermediate, unspecified substrings can be skipped, left unanalysed during one part of the parse and possibly reordered by the rule's application for later analysis by other rules. For instance, the GG rule:

$$A,\ gap(X),\ B,\ gap(Y),\ C \rightarrow$$
$$gap(Y),\ C,\ B,\ gap(X)$$

can be applied successfully to either of the following strings:

$$A,\ E,\ F,\ B,\ D,\ C$$

with gaps $X = E\ F$ and $Y = D$, and

$$A,\ B,\ D,\ E,\ F,\ C$$

with gaps $X = []$ and $Y = D\ E\ F$. Application of the rule yields

$$D\ C\ B\ E\ F$$

and

$$D\ E\ F\ C\ B$$

respectively. We can therefore think of the above GG rule as a shorthand for, among others, the two rules:

$$A,\ E,\ F,\ B,\ D,\ C \rightarrow D,\ C,\ B,\ E,\ F$$
$$A,\ B,\ D,\ E,\ F,\ C \rightarrow D,\ E,\ F,\ C,\ B$$

The idea of gapping grammars, as well as of the compiler implementation scheme shown below in Section 3.1 was developed in 1981 by V. Dahl as a result of examining Fernando Pereira's work on Extraposition Grammars, and finding the formalism limited, mainly with respect to the problem of treating coordinated constructs. During a 1982 visit to the University of Kentucky by V. Dahl, these ideas were tested in joint work with Michael McCord, but were later suspended in favour of a more promising approach to the coordination problem (see [Dahl&McCord,to appear]). We (Dahl & Abramson) now resume this research, no longer with a view towards treating coordination, but because the formalism itself has some rather interesting aspects.

Gapping grammars are interesting in the first place because each meta-rule, somewhat like a restricted version of VanWijngarden's two-level grammars which were used in the definition of Algol 68 [VanWijngarden,1975], represents infinitely many specific rules: each gap can be satisfied by many strings of terminals; to specify each of these unstructured substrings might require infinitely many grammar rules in other formalisms. Gapping grammars therefore cover a wide variety of rewriting situations using very few rules.

Secondly, there seems to be some psychological basis to the idea of focusing on the next relevant substring during analysis and leaving an intermediate one suspended in the background of consciousness, to be brought back into the focus of attention later, possibly repositioned with other more closely related substrings. When parsing discontinuous constituents, for instance as in the course and colloquial sentences "Desmond knocked the girl with green eyes down" as opposed to "Desmond knocked the girl with green eyes up", the human hearer will probably suspend his attention from the intermediate string "the girl with green eyes" until the completing substring to "Desmond knocked", i.e., "down" or "up", is heard, repositioned, and comprehended within its interrupted context.

A third argument for sometimes not specifying which constituents should be intermediate between two substrings is the fact that there is some empirical linguistic evidence in support of the existence of categories intermediate between lexical and phrasal categories [Radford,1981]. While these aren't clearly captured as traditional categories in linguistic theory, it is possible to computationally account for them simply by perceiving and naming them as gaps.

2. Background, Motivation, and Definition of Gapping Grammars.

Logic grammars originated with A. Colmerauer's Prolog implementation of Metamorphosis Grammars as an alternative notation for logic programs. They consist of rewriting rules where the non-terminal symbols may have arguments, and rule applica-

tion may therefore involve unification. For instance, a rule such as:

$$np(X) \rightarrow name(X)$$

can be applied to the strings $np(4)$ and $np(anne)$ yielding, respectively, $name(4)$ and $name(anne)$ but cannot be applied to either of the strings np or $np(z,y)$. The left hand side of a normalized Metamorphosis Grammar rule must start with a non-terminal symbol, but may be followed by a sequence of terminals (terminal symbols are written between / and /), whereas the right hand side may contain any sequence of terminals and non-terminals, as in:

$$a, [b], [c] \rightarrow [b], a, [c]$$

(Unnormalized Metamorphosis Grammars may contain rules beginning with a terminal, followed possibly by other terminals and non-terminals; there is no loss of generality, however, in restricting oneself to normalized MGs. See [Colmerauer,1978].) Definite Clause Grammars, DCGs, are a simplification of MGs in that rules are allowed only a single non-terminal on the left hand side, as in:

$$verb\_phrase(X) \rightarrow verb(X,Y),\ object(Y)$$

Extraposition Grammars (XGs) allow the interspersing of gaps in the left hand side,and these are routinely rewritten in their sequential order at the rightmost end of the rule, as in:

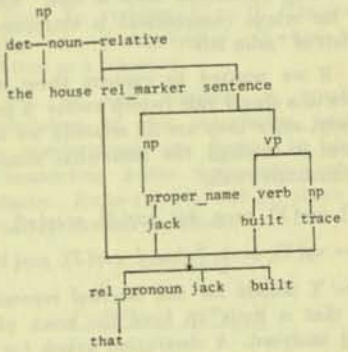
$$rel\_marker,\ gap(X),\ trace \rightarrow$$
$$rel\_pronoun,\ gap(X)^1 \qquad (1)$$

In an XG rule, symbols on the left hand side following gaps represent left-extraposed elements (e.g., "trace" above marks the position out of which the "noun_phrase" category is being moved in the relativization process).

Let us briefly examine the step-by-step rewriting of a sentence with a relative clause to understand how the gapping rule above works. Our complete grammar is:

```
sentence -> np, vp

np -> proper_name

np -> det, noun, relative
np -> trace

vp -> verb, np
vp -> verb

relative -> []
relative -> rel_marker, sentence

rel_marker,gap(X),trace ->
     rel_pronoun,gap(X)

det -> [the]

noun -> [house]

rel_pronoun -> [that]

proper_name -> [jack]

verb -> [built]
```

Applying these rules as graphed below, we analyse "the house that jack built" from *np:*



where the gap is "jack built". Notice that by adding appropriate symbol arguments to the rules, we can carry the antecedent's representation all the way to the constituent from which it was moved. Also notice that the same grammar, but with a larger lexicon, serves to analyse, for example, the sentence "the women who built the house", this time with an empty gap, and with the *trace* derived from the first *np* in the relative sentence.

Thus, XGs allow us to describe left-extraposition phenomena powerfully and concisely, and to arrange for the desired representations to be carried on to the positions from which something has been extraposed.

[1] We use our notation for consistency. Pereira's notation for $gap(X)$ is written *...* in the left hand side and simply left implicit on the right.

2.1. Motivation.

2.1.1. Left extraposition with more than one gap.

While XGs have the expressive power just shown, the restriction on how gaps are rearranged poses some expressive constraints even within the framework of left-extraposition. Consider for instance the noun phrase:

the man with whose mother john left

We can consider this noun phrase as the result of left-extraposing two substrings from:

the man | john left with [the] mother [of the man]|

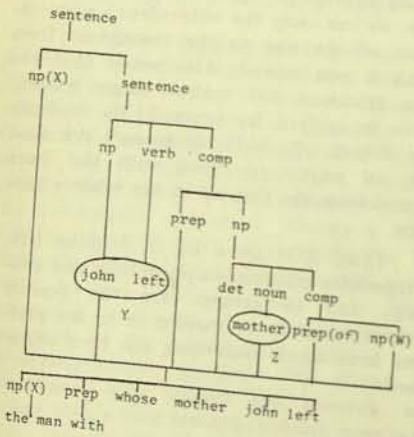
where "of the man" is left-extraposed before "the", and subsumed with it into "whose", and the whole complement is extraposed to the left of "john left".

If we wanted to capture these movements in a single rule (which seems a practical way, since they are all related), we might express it through the somewhat simplistic but illustrative rule:

np(X), gap(Y), prep, det, gap(Z), prep(of), np(X)

→ *np(X), prep, [whose], gap(Z), gap(Y)*

where X stands for the internal representation that is built up from the noun phrase being analysed. A derivation graph for this example would look roughly like:



Notice that the gapping rule's application unifies the internal representation X for "the man" with the representation W of the rightmost complement. The result of one partial analysis thus spreads to cover all implicit occurrences of the same substring.

2.1.2. Equivalent, preferred gapping formulations.

Fernando Pereira gives the following XG for the language $\{a^n b^n c^n\}$:

2.1.2.1. Grammar 1.

s -> as, bs, cs.

as -> [].
as, gap(X), xb -> [a], as, gap(X).

bs -> [].
bs, gap(X), xc -> xb, [b], bs, gap(X).

cs -> [].
cs -> xc, [c], cs.

Other formulations of grammars which use gaps are conceivable, however, and it should be up to the grammar writer to choose a formulation unconstrained by fixed reordering rules. The following GG, for example, describes the same language:

2.1.2.2. Grammar 2.

s -> as, bs, cs.

as -> [].
as -> xa, [a], as.

bs -> [].

xa, gap(X), bs -> gap(X), [b], bs, xb.

cs -> [].

xb, gap(X), cs -> gap(X), [c], cs.

In Grammar 1, symbols such as *xb* can be considered as marks for *b*'s which are being left-extraposed. In Grammar 2, such marks can be seen as right-extraposed. Whereas in this particular example our choice may just be a matter of personal preference, there are cases in which movement is more naturally thought of in terms of right rather than left-extraposition (as in "The man is here that I told you about"). There also may be efficiency reasons to prefer a right-extraposing formulation: in the first implementation below of GGs, Grammar 2 above works faster than Grammar 1.

2.1.3. Interaction between different gapping rules.

Consider the language $\{a^n b^m c^n d^m\}$ which can be described by the following GG:

```
s -> as, bs, cs, ds.

as -> [].
as, gap(X), xc -> [a], as, gap(X).

bs -> [].
bs, gap(X), xd -> [b], bs, gap(X).
cs -> [].
cs -> xc, [c], cs.

ds -> [].
ds -> xd, [d], ds.
```

This is a perfectly good GG. XGs cannot, however, be used in this situation because of the XG constraint on the nesting of gaps: two gaps must either be independent, or one gap must lie entirely within the other.

3. Implementations of GGs.

3.1. A Compiler: beautiful but dumb.

Typically, logic grammars are translated into Prolog programs by augmenting each non-terminal symbol with two arguments: one argument X which represents the input string yet to be parsed, and the other argument Y which represents what is left of the input string after the rule being applied has consumed some of it. We then say that the string $X - Y$ (read as "X minus Y") can be recognized as belonging to the category denoted by the non-terminal. Thus, a rule such as:

$$sentence \rightarrow name,\ verb.$$

is translated into a Prolog clause:

$$sentence(X_1,X_3) \leftarrow$$
$$name(X_1,X_2),\ verb(X_2,X_3) \qquad (a)$$

meaning roughly: "there is a sentence in the string $X_1 - X_3$ if there is an initial substring $X_1 - X_2$ that can be parsed as a *name* and is followed by a substring $X_2 - X_3$ that can be parsed as a *verb*".

Terminal symbols do not give rise to Prolog predicates, but become instead involved in the specification of the input and output strings being manipulated by the non-terminals. For instance, the rules

$$name \rightarrow [mary].$$
$$verb \rightarrow [laughs].$$

can be translated into the unit clauses:

$$name([mary|X],X) \leftarrow \qquad (b)$$
$$verb([laughs|X],X) \leftarrow \qquad (c)$$

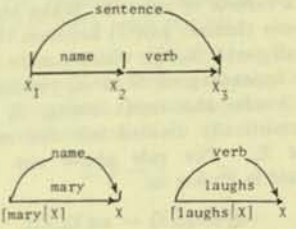
where (*b*) means roughly: "a *name* is recognized in any input string which begins with 'mary', yielding an output string which is the remainder of the input string after consuming 'mary'".

Thus, with respect to rules (*a*), (*b*), and (*c*), a query such as:

$$sentence([mary,laughs],[])$$

will unify X_1 with *[mary,laughs]* and X_3 with *[]*, proceed to consume a name from X_1, yielding $X_2 = [laughs]$, and then consuming a verb from X_2, yielding $X_3 = []$. The string $X_1 - X_3$, i.e., *[mary,laughs] - []*, has been recognized as a sentence.

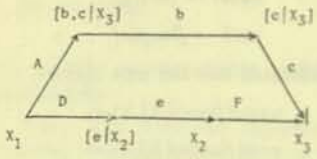
Let us now consider a graphical representation of this translation process, where non-terminals are viewed as labeled arcs connecting nodes representing phrase boundaries. Rules (*a*), (*b*), and (*c*) above can then be represented as follows:



Normalized MG rules accept a sequence of terminals after the single non-terminal head on the left hand side (since more than one non-terminal would result in a non-Horn clause). The translation of such a rule to Prolog may be represented graphically by adding more arcs to the upper part of the graph. The rule

$$A,\ [b],\ [c] \rightarrow D,\ [e],\ F$$

would translate as indicated by:



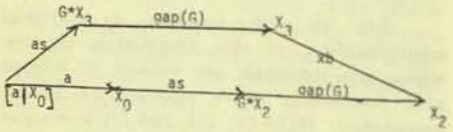
which is the Prolog clause:

$$A(X_1,[b,c|X_3]) \leftarrow D(X_1,[e|X_2]),\ F[X_2,X_3).$$

Let us now consider a rule with gaps and how it should be represented graphically:

$$as,\ gap(G),\ zb \rightarrow [a],\ as,\ gap(G)$$

We can think of a gap G simply as a substring of the input that is skipped unanalyzed and appended elsewhere in the output string. Thus, if we denote the appending of G to a string z as $G*z$, we can represent this rule graphically by:



The symbol $gap(G)$, in fact, can be thought of as a version of *append*. When translating rules into clauses, $gap(G)$ becomes the predicate call $gap(G,X_1,X_0)$, which can be specified as the appending of G to X_0 yielding X_1. In other words, the input string X_1 is nondeterministically divided into two substrings G and X_0. The rule above can thus be expressed in Prolog as:

$$as([a|X_0],X) \leftarrow as(X_0,X_1),$$
$$gap(G,X_1,X_2),\ gap(G,X,X_3),\ zb(X_3,X_2)$$

or alternatively as:

$$as([a|X_0],X) \leftarrow as(X_0,X_1),\ append(G,X_{2,X_1}),$$
$$append(G,X_3,X),\ zb(X_3,X_2)$$

Notice that the remainder of the rule's left hand side becomes a part of the clauses's body, and we therefore remain within the Horn clause subset of first order logic. Notice too, that this translation scheme allows the left hand side of a rule to be nearly as unrestricted as the right hand side: though the head must be a non-terminal, any combination of terminals, non-terminals, and even Prolog calls can be expressed as context.

The compiler shown below produces the corresponding Prolog clauses from gapping grammar rules by first constructing two pseudo-rules. From a rule such as

$$A,\ B \rightarrow C$$

where A is the non-terminal head symbol, and where B is the remainder of the left hand side of the rule, and C is the right hand side, it constructs clauses corresponding to the pseudo-rules:

$$c\_nonterm \rightarrow C$$
$$b\_nonterm \rightarrow B$$

where *c_nonterm* and *b_nonterm* are pseudo-non-terminals. In doing so, it also binds the output strings corresponding to both pseudo-non-terminals. In our example, the clauses generated are:

$$c\_nonterm([a|X_0],Z) \leftarrow as(X_0,X_1),\ gap(G,X_1,Z)$$
$$b\_nonterm(X,Z) \leftarrow gap(G,X,X_2),\ zb(X_2,Z)$$

Next it constructs the head of the desired clause by using and retrieving the input and output strings from the input strings of *c_nonterm* and *b_nonterm*. In our example this yields

$$as([a|X_0],X)$$

The desired clause's body is constructed by appending the two bodies of the pseudo-clauses

$$as([a|X_0],X) \leftarrow as(X_0,X_1),$$
$$gap(G,X_1,X_2),\ gap(G,X,X_3),zb(X_3,X_2)$$

The compiler's full listing is shown below. In addition to accepting purely syntactic gapping grammar rules, it also accepts gapping grammar rules with a superadded *semantic* component to specify a translation. The general form of such a rule is:

$$A,\ B \rightarrow C<:>Sem$$

where *Sem* consists of one or more Horn clauses which specify how semantic attributes of the head symbol A are computed in terms of semantic attributes of C and possibly even of B. The Horn clauses in *Sem* govern traversal of the derivation or parse tree which is constructed automatically [Abramson,1984]. Gapping grammar rules which are

purely syntactic have the trivial semantic unit clause *true* attached to them. The predicate *form_node* below creates the derivation tree for the head symbol A by concatenating the trees for the pseudo-clauses corresponding to B and C.

```
synal((A,B -> C<:>Sem),Clause) :- !,
 expand_term(
   (c_nonterm->C<:>Sem),CClause),
 expand_term((b_nonterm->B),BClause),
 clauseparts(CClause,CHead,CBody),
 clauseparts(BClause,BHead,BBody),
 CHead =.. [c_nonterm,CTree,X,Z],
 BHead =.. [b_nonterm,BTree,Y,Z],
 A     =.. [Pred|Args],
 form_node(CTree,BTree,Pred,ATree),
 concaten(Args,[ATree,X,Y],NewArgs),
 NewA  =.. [Pred|NewArgs],
 combine(CBody,BBody,Body),
 formclause(NewA,Body,Clause).

synal((A,B -> C),Clause) :- !,
  synal((A,B -> C<:>true),Clause).

clauseparts((Head :- Body),Head,Body) :- !.
clauseparts(Head,Head,true).

formclause(Head,true,Head) :- !.
formclause(Head,Body,(Head :- Body)).

combine(true,B,B) :- !.
combine(A,true,A) :- !.
combine(A,B,(A,B)).

form_node(node(_,N1,Sem),
       node(_,N2,_),
       Pred,node(Pred,N,Sem)) :-
       node(Pred,N,Sem)) :-
  concaten(N1,N2,N).

gap([]) -> [].
gap([Word|List]) -> [Word], gap(List).

concaten([],X,X).
concaten([X|L],M,[X|N]) :- concaten(L,M,N).
```

The beauty of the compiler resides in its simplicity and conciseness. The compiler is dumb, however, in that the gap predicate successively consumes substrings of length 0, 1, 2,... with no further control than simple backtracking as to what should be in the gap. Thus, even on simple languages, such as $\{a^n b^n c^n\}$ with relatively low values of n, say $n = 5$, it is very slow. Some more information needs to be incorporated in the gap predicate, but this seems to involve dynamic information about the state of the computation, and such information is accessible only in some Prolog implementations. Another alternative which we are considering is to use concurrency in parsing; we sketch this idea below and are planning a future detailed article on the subject.

Although the ideas on compiling GGs are due to V. Dahl, credit is due to Michael McCord for the actual writing of the compiler in terms of pseudo-clauses.

3.2. Another Compiler: Efficient but not general.

In this section we introduce a class of Gapping Grammars which can be implemented in Prolog efficiently. This class consists of those Gapping Grammars in which each gapping rule is of the form:

$$\alpha,\ gap(X),\ /term/ \rightarrow \gamma,\ gap(X) \qquad \text{(A)}$$

That is, there is only one gap which is rewritten to the rightmost position of the right hand side, and on the left there is a single (pseudo-)terminal following the gap. This class of grammars includes a subclass of Pereira's Extraposition Grammars, but also, depending on the definition of the *gap* and *fill* predicates, may include grammars which cannot be handled by Extraposition Grammars, such as, for example, a grammar for the language $\{a^n b^m c^n d^m\}$, with $m,n \geq 0$.

This class may be viewed as a generalization of normalized Metamorphosis Grammars. A normalized Metamorphosis Grammar rule is of the form:

$$\alpha,\ \beta \rightarrow \gamma \qquad \text{(B)}$$

or

$$\alpha \rightarrow \gamma \qquad \text{(C)}$$

where

$$\alpha \in V_N$$

$$\beta \in V_T^*$$

and

$$\gamma \in V_T^* \bigcup V_N^*$$

The notation $gap(X),\ /term/$ therefore represents a large set of MG rules.

The implementation technique is based on message passing during parsing and rests on the following considerations. The terminal symbols which occur on the left hand side of

XG rules and to the immediate right of a gap may be said to be pseudo-symbols in that they are generally not expected to occur in input strings to be parsed, but are generated during parsing to act as signals of some sort, and are absorbed later in the parse. Consider, for example, in the XG grammar for the language $\{a^n b^n c^n\}$ the rule:

$$as,\ gap(X),\ xb \rightarrow [a],\ as,\ gap(X)$$

The *xb* is generated to mark the end of the gap and to count an occurrence of an *[a]*. The *xb* is then absorbed by a matching *[b]* in the rule:

$$bs,\ gap(X),\ xc \rightarrow xb,\ [b],\ bs$$

Similarly, in the XG for a small subset of English, the rule (1) in Section 2 generates *trace* to mark the point from which a noun phrase has been left-extraposed, and the rule

$$np \rightarrow trace$$

absorbs the *trace*. The introduction of such pseudo-symbols, moreover, produces a slight theoretical problem in that they may occur in some sentential forms of the grammar, but not in the terminal sentential forms.

Since there is only one gap in our restricted rules, and this gap is followed by a "terminal", we write instead of *A* the following:

$$\alpha,\ gap(X),\ [term] \rightarrow \gamma \qquad \text{(D)}$$

and read this: an α, in the context of a gap which is terminated by a signal *term* may be rewritten to a γ followed by the gap. The gap is implicit on the right hand side of the rule. Thus our signal gapping grammar rules are of the form (B), (C) or (D). The sending of a signal which closes such a gap is indicated by the predicate

$$fill(term)$$

which generates (accepts) the empty string. Our version of the grammar for the language $\{a^n b^n c^n\}$ is as follows:

```
s->as,bs,cs.

as,gap(X),[xb]->[a],as.
as->[].

bs,gap(X),[xc]->fill(xb),[b],bs.
bs->[].

cs->fill(xc),[c],cs.
```

In implementing this form of GG we specialize the *synal* predicate as follows:

```
synal(((A,gap(Name),[Signal])->C<:>Sem),
  Clause) :- !,
 expand_term(
  (c_nonterm->
  (C,gap(Signal,Name))<:>Sem),
  CClause),
 clauseparts(CClause,CHead,CBody),
 CHead =.. [c_nonterm,G0,Gn,CTree,X,Z],
 A     =.. [Pred|Args],
 GTree = node(gap,[Signal,Name],true),
 form_node(CTree,GTree,Pred,ATree),
 append(Args,[G0,Gn,ATree,X,Y],NewArgs),
 NewA  =.. [Pred|NewArgs],
 combine(CBody,
       gap(Signal,Name,GTree,Y,Z),
       Body),
 formclause(NewA,Body,Clause).

synal(((A,gap(Name),[Signal]) -> C),
  Clause) :- !,
 synal(
  ((A,gap(Name),[Signal]) -> C<:>true),
  Clause).
```

In the goal *expand_term* the *Signal* is added to the named gap which is placed at the right end of the syntactic portion of the rule; since the only context of a rule is of the form *gap(Name), [Signal]*, we dispense with *BClause* and construct the clause for *A* directly; other changes in *form_node* involve the formation of a "tree" to record the contents of the gap as a difference list (see below). *synal* compiles, for example, the rule

$$bs,\ gap(Name),\ [xc] \rightarrow fill(xb),\ [b],\ bs.$$

to:

```
bs(S0,
  S3,
  node(bs,
     [FillTree,
      b,
      BsTree,
      GapTree,
      node(gap,[xc,Name],true)],
     true),
 X,
 Y) :-
fill(xb,S0,S1,FillTree,X,X1),
```

```
c(X1,b,X2),
bs(S1,S2,BsTree,X2,X3),
gap(xc,Name,S2,S3,GapTree,X3,Z),
gap(xc,Name,node(gap,[xc,Name],true),Y,Z).
```

The reader will notice that in addition to the pair of arguments for the "input" and "output" strings (*X,X1,X2,X3,Y,Z*), and the argument for the parse tree, there is another pair of arguments - the "input message stream" and the "output message stream" - which has been added to all the non-terminals except the rightmost instance of *gap*. These are *S0*, *S1*, *S2*, and *S3*, and are added to non-terminal symbols only by the predicate *translate_rule* (not shown here, but called by *expand_term*: see [Abramson,1984]) which processes non-gapping rules. Note that non-gapping rules are normalized metamorphosis grammar rules and are translated as outlined in Section 3.1. The ordinary non-terminals, such as *bs*, will neither add messages to the input stream nor delete messages from the input stream in order to produce a new output stream: the input stream will be passed to whatever is called, and a possibly new output stream will be formed as a result of the call. Messages are inserted by *gap* and removed by *fill*. Let us examine the definition of *gap* and *fill* to see how these streams are manipulated:

```
gap(Symbol,
    Gap,
    node(gap,[Symbol,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

gap(Symbol,
    Gap,
    StackIn,
    [[Symbol,Gap]|StackIn],
    node(gap,[Symbol,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

fill(Symbol,
    [[Symbol,Gap]|StackOut],
    StackOut,
    node(fill,[Symbol,Gap],true),
    EndGap,
    EndGap) :-
      Gap = StartGap - EndGap.
```

When *gap* is called with a pair of stream arguments, the start of a gap is known. *Gap* is instantiated to the difference list *StartGap* – *EndGap*, with *EndGap* uninstantiated. The pair *[Symbol,Gap]* is added to the input message stream to form a new output message stream. The *Symbol* is the signal which will indicate the end of a gap. When *gap* is called without the stream arguments, as in the last call to gap in the compiled version of *bs*, the context is merely being checked (please refer to the discussion of *synal* in the previous section) and the input and output strings, *StartGap* and *EndGap*, respectively, verify the extent of the gap. *EndGap* will still be uninstantiated.

When *fill* is called, the end of a gap with the signal *Symbol* has been found. There must be a pair of the form *[Symbol,Gap]* at the front of the input message stream. *EndGap* is instantiated at this point, and the pair is removed from the input message stream to yield a new output message stream. When *EndGap* is instantiated, the "trees" of the *gap* and *fill* predicates, which have been made to look like ordinary non-terminals, are also instantiated. The trees for both *gap* and *fill* contain a record of the signal *Symbol* and the gap itself as the difference list to which *Gap* is instantiated. The message streams act as a stacking mechanism for unfilled gaps. Note that *fill* accepts the empty string.

A string is parsed with this grammar by a call to:

```
s(Source) :-
    s([],[],Tree,Source,[]).
```

which indicates that *Source* is an *s*, with no input left, and that no messages are left in the streams, ie, the stack of messages, initially empty, is empty at the end of parsing. A parse tree *Tree* records the derivation. (See [Abramson,1984]).

With this definition of *gap* and *fill* we have a new implementation of a subset of XGs: it contains rules with only one gap followed by a terminal. The compiler for this subset, *synal* above, is somewhat simpler than the general processor of Pereira.

By changing the definition of *gap* and *fill*, however, we can process grammars which cannot be handled by XGs. Here is our signalling GG for the language $\{a^n b^m c^n d^m\}$:

```
s->as,bs,cs,ds.

as,gap(X),[xc]->[a],as.
as->[].

bs,gap(X),[xd]->[b],bs.
bs->[].

cs->fill(xc),[c],cs.
cs->[].

ds->fill(xd),[d],ds.
ds->[].
```

We redefine *gap* and *fill* so that the input and output message streams manipulate a pair of stacks, one to handle *xc* signals, and the other to handle *xd* signals. The gaps can now be dealt with independently of one another.

```
gap(Symbol,
    Gap,
    node(gap,[Symbol,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

gap(xc,
    Gap,
    [StackC,StackD],
    [[[xc,Gap]|StackC],StackD],
    node(gap,[xc,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

gap(xd,
    Gap,
    [StackC,StackD],
    [StackC,[[xd,Gap]|StackD]],
    node(gap,[xd,Gap],true),
    StartGap,
    EndGap) :-
      Gap = StartGap - EndGap.

fill(xc,
    [[[xc,Gap]|StackC],StackD],
    [StackC,StackD],
    node(fill,[xc,Gap],true),
    EndGap,
    EndGap) :-
      Gap = StartGap - EndGap.

fill(xd,
    [StackC,[[xd,Gap]|StackD]],
    [StackC,StackD],
    node(fill,[xd,Gap],true),
    EndGap,
    EndGap) :-
      Gap = StartGap - EndGap.
```

The general GG implementation is very powerful and inefficient; this implementation, although not general, is more efficient; and is, at the cost of some programming of the *gap* and *fill* predicates by the grammar writer, extendable to classes of grammars with independent gapping systems which cannot be handled by XGs. It is interesting that subclasses of GGs can be parameterized by data structures: one may think of trying to characterise the subclass of GGs with a queue (deque, tree) implementation of *gap* and *fill*, for example.

A complete listing of these Prolog implementations is available from H. Abramson or V. Dahl.

3.3. Towards a concurrent implementation of gapping grammars.

The beautiful but dumb compiler is inefficient because of the way it tries to establish what is contained in a gap. It simulates the non-deterministic breaking up of the input string into the contents of the gap and the unconsumed output string by trying one solution of *append(Gap,Output,Input)*, backtracking to the next solution if the first is not suitable, and so on. A concurrent implementation might, however, proceed as follows. For each solution of *append(Gap,Output,Input)* a copy of the process which represents the state of the parse so far is created. Each of these processes is a clone of the original process up to the call of *gap*. Each process continues, however, with a different solution to *append(Gap,Output,Input)*. Those processes which have not been given a solution which will permit the parse to continue will eventually die. Those processes which have been given a solution which allows the parse to complete will each be left suspended at the end with a derivation tree representing the successful parse. (Note that this notion of process is similar to the notion of process which is used in the Unix operating system.) For this strategy to work, it will be necessary to have a meta-logical predicate which gives access to the state of a Prolog computation. This strategy utilizes independent sequential Prolog processes - the parsing, except when handling a gap, proceeds by top-down, depth-first search with backtracking. An

alternative strategy would be to develop an entirely concurrent implementation of grammars.

The authors plan to investigate whether Concurrent Prolog [Shapiro,1983], the distributed logic of [Monteiro,1982], or Epilog [Pereira,1982], [Porto,1982] could easily specify such implementations of Gapping Grammars.

4. Discussion, work in progress.

4.1. Advantages of gapping grammars.

GGs, although theoretically no more powerful than MGs - which have the computational power of a Turing machine - have more expressive power than MGs in that they permit the specification of rewriting transformations involving components of a string separated by arbitrary strings. The expressive power takes the form of conciseness: one does not have to give a rule or rules for the generation of the intervening string, but rather a single meta-rule involving gaps replaces a possibly infinite set of non-gapping rules.

One aspect of GG expressiveness has not yet been fully explored. GGs, like MGs and XGs, allow Prolog calls in the right hand side of a rule, but unlike them, GGs allow Prolog calls in the left hand side of a rule (refer to *synal* above to see why this is so). It is possible therefore to write GGs which can establish context checks dynamically during parsing.

The compiler for GGs - our second implementation - provides an alternative implementation of a restricted class of extraposition grammars, but also, depending on the definition of *gap* and *fill*, provides the grammar writer with a mechanism for writing rules which go beyond the nesting constraints of the XG formalism. Our example above shows how to deal with two independent gapping systems: the extension to the general case is obvious. Another possibility is to parameterize classes of grammars by the data structures used to implement the *gap* and *fill* predicates, for example, by queues instead of stacks, etc. Another extension lies in permitting the signal to be parameterized, i.e., instead of having rules of the form (*D*) with *term* a functor of zero arity, *term* might be a functor of positive arity. This would permit more sophisticated gap handling by the *gap* and *fill* predicates.

4.2. Limitations.

In some cases GGs may prove, however, to be too powerful. Consider, for instance, the following grammar which one naively might think suitable for checking that input strings are balanced with respect to (and):

```
left, gap(X), [')'] -> ['('], gap(X).

s -> left, [')'], gap(X), s.
s -> [].
```

With this grammar, strings such as $(a + (b + c))$ and $((a + b) - (c * d)) / f$ are recognized as balanced, but also a string such as $(a + b$ is recognized as balanced. The reason for this error is that nothing in the grammar precludes the gaps from containing parentheses, so that unbalanced parentheses will be absorbed into gaps. The grammar can, of course, be modified so that only those strings which are balanced with respect to parentheses are accepted, but it seems appropriate for the grammar formalism to provide the user with a convenient means for constraining the gaps. It would be interesting to determine how much of an extension along these lines could be usefully provided without falling into the trap of describing the complement of a language.

Another approach to be investigated with respect to too general a notion of gaps is allowing strings *not* in the language to be generated, these strings to be subsequently filtered out by another process. Primitives for describing filters would then be necessary. In natural language applications, a mixture of both approaches may be needed. Both constraints and filters have already been proposed in Chomsky's Extended Standard Theory (see references in [Radford,1982]), and, it would be interesting to study ways of constraining and filtering GG rules in the light of this theory.

4.3. Work in progress.

We have only tentatively sketched a concurrent implementation of GGs. Details of this strategy have to be worked out and specified in Prolog, Concurrent Prolog [Shapiro,1983], the distributed logic of [Mon-

teiro,1982], or Epilog [Pereira,1982], [Porto,1982]. Ideally, a parallel architecture should support a concurrent GG system.

Another implementation of GGs which we are exploring is an interpreter which works with derivations directly rather than with Prolog calls of non-terminal procedures. By this method, we would for a rule such as

$$s \rightarrow as, bs, cs$$

rather than call as, then bs, then cs, maintain a list of goals which would represent a sentential form. The original list of goals s would be replaced by a list of goals as, bs, cs. Context sensitive rules would involve manipulation of the goals in the sentential form to see if some of them could appropriately derive the desired context.

These extensions, as well as the addition of constraints and filters to GGs, are the object of future research by the authors.

5. Acknowledgements.

This work was supported by the National Science and Engineering Research Council of Canada. Michael McCord's contribution, mentioned in Section 3.1, is gratefully acknowledged. A referee's suggestion that we expand on our comments about right extraposition will be treated in a later paper: length constraints prevent such expansion here.

6. References.

Abramson, H., Definite Clause Translation Grammars Proceedings IEEE Logic Programming Symposium, 6-9 February 1984, Atlantic City, New Joisey.

Colmerauer, A., Metamorphosis Grammars, in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

Dahl, V. Translating Spanish into logic through logic, American Journal of Computational Linguistics, vol. 13, pp. 149-164, 1981.

Dahl, V. & McCord, M. Treating Coordination in Logic Grammars, to appear in American Journal of Computational Linguistics.

Monteiro, L., A Horn clause-like logic for specifying concurrency, Proceedings of the First International Logic Programming Conference, pp. 1-8, 1982.

Pereira, F.C.N., Extraposition Grammars, American Journal of Computational Linguistics, vol. 7 no. 4, 1981, pp. 243-255.

Pereira, L.M., Logic control with logic, Proceedings of the First International Logic Programming Conference, pp. 9-18,1982

Pereira, F.C.N. & Warren, D.H.D, Definite Clause Grammars for Language Analysis, Artificial Intelligence, vol. 13, pp. 231-278, 1980.

Porto, A. Epilog - a language for extended programming in logic, Proceedings of the First International Logic Programming Conference, pp. 31-37, 1982.

Radford, A., Transformational Syntax Cambridge University Press, 1981.

Shapiro, E.Y., A subset of Concurrent Prolog and its interpreter, ICOT Technical Report TR-003, 1983.

Van Wijngarden, A. et al., Revised report on the algorithmic language Algol 68, Acta Informatica, vol. 5, pp. 1-236, 1975.

Eager and Lazy Enumerations in Concurrent Prolog

Hideki Hirakawa, Takashi Chikayama, Koichi Furukawa

ICOT Research Center
Institute for New Generation Computer Technology,
Mita Kokusai Bldg. 21F, 4 - 28 Mita 1-chome,
Minato-ku, Tokyo 108

ABSTRACT

Logic programming languages have inherent possibility for AND-parallel and OR-parallel executions. Concurrent Prolog designed by E.Shapiro introduces an AND-parallelism and an limited OR-parallelism, i.e, a don't-care-nondeterminism. The other aspect of OR-parallel execution, i.e, don't-know-nondeterminism is formalized as an 'eager_enumerate' operation on a set expression. This paper describes a computational model which provides the eager enumerate function to Concurrent Prolog and shows its implementation in Concurrent Prolog itself. This paper also shows a lazy enumerate function can be implemented easily by introducing a bounded buffer communication technique to the eager enumerator.

1. INTRODUCTION

A growing area of research in highly parallel processing covers computer architectures, programming languages and computational models. One of the best candidates for a high level machine language for highly parallel processors is a logic programming language which represents AND and OR relations between predicates. Logic programming languages possess inherent potential for parallel processing, that is, AND-parallel and OR-parallel execution.

Based on this concept, several parallel programming languages have been proposed: such as KL1 (Furukawa et al. 84), Concurrent Prolog (Shapiro 83), PARLOG (Clark and Gregory 83) and Bagel machine language (Shapiro 84). Researches in parallel programming are being conducted using these languages. In these languages, AND-parallelism is used for the description of parallel processes, which is based on the process interpretation of logic (Emden 82). OR-parallelism has two aspects, the so-called don't-care-nondeterminism and don't-know-nondeterminism (Kowalski 79). The don't-care-nondeterminism is adopted in all the languages mentioned above. However, the don't-know-nondeterminism is introduced only in PARLOG and KL1 where it is used to find multiple solutions for a query. PARLOG and KL1 use a "set expression" as the interface between AND-parallelism and OR-parallelism (don't-know-nondeterminism).

In this paper we regard the OR-parallelism for finding all solutions as enumerating elements of a set in the same way as in KL1. This paper describes the 'enumeration' in Concurrent Prolog, which the implementation of the OR-parallel execution in the AND-parallel execution. An advantage of this approach is that both AND-parallel and OR-parallel

execution can be achieved within a small basic framework of Concurrent Prolog. This implies a decrease in the complexity of the architecture and in the amount of hardware required in the parallel machine.

Various models for parallel processing of logic programs are proposed from the computational model viewpoints. Nitta and Conery described parallel interpretation methods based on an AND/OR process model (Nitta et al. 83), (Conery 83). Haridi proposed a language based on natural deduction, which covers a wider class of statements than Horn Logic (Haridi and Sahlin 83). Hirakawa proposed a computational model based on multi-processing and graph reduction mechanism (Hirakawa et al. 83). In this paper, a computational model for Pure Prolog is introduced with the model based on multi-processing and message communication between processes. In this model, goals are computed serially from left to right, and clauses are computed in parallel.

Based on this model, we have implemented a Pure Prolog interpreter in DEC-20 Concurrent Prolog (Shapiro 83) to realize the 'enumerate' function mentioned above. This interpreter works eagerly to get all solutions for a given goal. By replacing the stream communication portion used in the interpreter with bounded buffer communication implementation and by adding some small changes, a lazy interpreter which works in accordance with demands can be easily obtained.

Section 2 of this paper explains the concept of 'enumeration'. Section 3 describes the computational model and its implementation in Concurrent Prolog. Section 4 describes the modification of the interpreter from the eager version to the lazy one.

2. ENUMERATIONS

An interface between AND-parallelism and OR-parallelism (a don't-know-nondeterminism) is introduced using set expressions in PARLOG and KL1. A set expression has the syntax such as:

{X|Y} where X is a term and Y is a goal sequence

In KL1, the basic operation on a set is an 'enumerate' operation. In this paper the same expression is introduced in Concurrent Prolog as in KL1. 'Enumerate' is similar to the 'bagof' operation in DEC-10 Prolog (Warren 81).

Prolog : bagof(X,Y,Collection)
Concurrent Prolog :
enumerate({X|Y},Stream)

The meaning of the 'bagof' literal above is "Collection is the collection of terms of the form X, which satisfy the goal sequence Y". In Concurrent Prolog, 'Stream' in the 'enumerate' clause is the same as 'Collection' in 'bagof' logically, but it is a stream of terms rather than a simple collection. This is a natural interface to an AND-parallel process.

There are two types of streams. One is an uncontrolled stream and the other is a controlled stream. 'Uncontrolled' means that once 'enumerate' is called, its output stream is never stopped until all the solutions are generated. On the other hand, 'controlled' means that the generation of the solutions is invoked by a demand of a process outside of 'enumerate'. The former type of enumeration is called 'eager enumeration' and the

latter 'lazy enumeration'. The eager enumeration is used for finding all solutions to a database query and generally requires many computation resources, while lazy enumeration is used for finding a part of the solutions which satisfy some requirements of other processes. The eager and lazy computation mode for the 'collection' is introdeced as primitives for the control of logic programming (Kahn 84). The following are simple examples of lazy and eager enumerations.

```
Eager enumeration : "display all
  countries with a population of
  more than one hundred million"

Goal: eager_enumerate(
       {Nam|country(Nam,Capl,Pop),
            Pop>100},Str),
      display_stream(Str?).

Lazy enumeration : "display three
  countries with a population of
  more than one hundred million"

Goal: lazy_enumerate(
       {Nam|country(Nam,Cap,Pop),
            Pop>100},Str?),
      display(Str,3).
```

In the above examples, 'enumerate' and 'display' run in parallel (concurrently). In the former example, 'eager_enumerate' produces a stream of country names and 'display_stream' displays them in turn. In the latter example, 'display' sends three demands for solutions to 'lazy_enumerate' and 'lazy_enumerate' produces them.

3. EAGER ENUMERATION

The eager enumeration is provided by a Prolog interpreter which computes subgoals serially and clauses in parallel. In this section, a computational model for an eager interpreter and its implementation in Concurrent Prolog are described.

3.1 Computational Model

3.1.1 Components

The computational model for the eager interpreter consists of three components: processes, channels and a Horn Clause Database (HDB).

A process plays a key role in a computation. An arbitrary number of processes can be generated in a system. A process corresponds to a clause being computed, such as H<--G1,G2. There are two types of processes, that is, active and waiting. The waiting process waits until it receives data from another process.

A channel is a communication path between processes and is dynamically generated during the computation. Data transferred through a channel is called a message. A message is passed from a process called a "generator" to processes named "consumers". The distinction between a generator and a consumer is relative, and a single process can simultaneously play both roles. One generator process can simultaneously send a message to multiple consumer processes via a channel. Similarly, one consumer process can be connected to multiple generators.

The Horn Database (HDB) is a set of Pure Prolog clauses. A process can fetch a set of clauses which have heads unifiable with a certain term. A fetching operation about term P is called a "P-related fetch".

3.1.2 Process Operation

In the computational model given here, computation progresses while multiple processes are ex-

changing messages. This subsection provides a more detailed description of the process, shows a simple example, and presents the execution mechanism of the computational model.

A process is defined by five components: Status, Head, Goals, Input-Channel, and Output-Channel, as shown in the following format:

process(Status,Head,Goals,IC,OC)

'Status' indicates the state of a process and is either 'active' or 'waiting'. 'Head' is a predicate (term) and represents what the process must eventually compute. 'Goals' is either null, 'true' or a sequence of predicates and indicates the predicates to be computed to compute the Head. For example, if the HDB includes 'a<--b,c', there may be the following process:

process(Status,a,(b,c),IC,OC)

In addition, if the predicate b has been computed, there may be a process as follows:

process(Status,a,(c),IC,OC)

A 'channel' is used to transfer messages among processes as described above. A process appears as a consumer for its input channel (IC), while it functions as a generator for its output channel (OC).

Here, we will define the operation of a process.

(A) Active process

The operation mode of an active process is either reduction or termination. In reduction mode, the leftmost subgoal of a clause is expanded using inference rules in HDB; the active process is maintained after the reduction is completed. By contrast, termination means that inference reaches 'true' or the application of an inference rule fails; in both cases, the process is immediately deleted.

Operation in reduction mode

Assume process(active,H,G,I,O). If G is neither null nor 'true' and G is in the form of either P or (P,...) where P is a predicate defined in the HDB, then the process performs a P-related fetch to the HDB to obtain a clause set, S, generates active processes for all the components of S, and connects each process with itself through Channel I (each process functions as a producer). It also changes its status to 'waiting'.

Operation in termination mode

There are two types of terminations: success or failure. A success termination occurs when reduction reaches true, while a failure termination occurs when a fetch operation fails. The failure termination corresponds to Prolog's 'fail'.

Success termination

When G is either null or true, the process sends H via channel

Failure termination

The process deletes itself.

(B) Waiting process

Having received a message (term) M via channel I, a waiting process generates G', a copy of its Goals G, in the format P' or (P', P1,...), and unifies the head element P' with M (transfer of the computation results) Then, it establishes NewG, which is G' with its head element removed. However, when G' contains only P', NewG will be true. Then, the waiting process generates the

following active process:

process(active, H, NewG, I', O)
Where I' is a new channel.

The waiting process will be maintained in the original form.

The entire computation terminates, when all the processes are deleted.

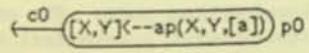
3.1.3 Computation Example

This subsection presents a simple example to show the way the computational model is executed. In the following figures, the active process p, the waiting process q and the channel c are denoted by (p), [q] and <—c—, respectively (p, q and c may be omitted). The Head H and Goals G are shown as H<--G.

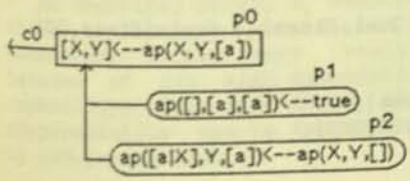
Assume that the HDB is given as follows:

{ ap([],X,X).
ap([U|X],Y,[U|Z])<--ap(X,Y,Z) }

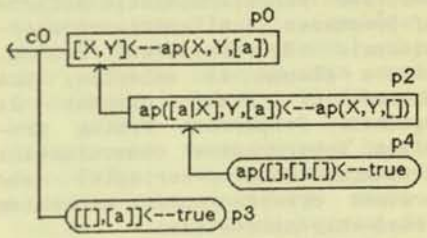
To compute [X,Y] that satisfies a goal ap(X,Y,[a]), the following process is generated as the initial process:



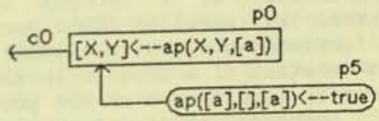
A message output through c0 is the solution. Since p0 is an active process, it performs a fetch operation and generates new processes, p1 and p2, and then changes its status from active to waiting.



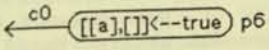
There are two active processes. Each process runs simultaneously. As p1 has a terminated clause, it sends the head of the clause and deletes itself; p0 receives message 'ap([],[a],[a])' and creates a new process p3; p2 performs a reduction mode operation and produces a new process p4.



An active process p4 sends the message 'ap([],[],[])' to p2 and deletes itself. Receiving the message, p2 creates a new process p5 and deletes itself because it has no child process; p3 sends '[[],[a]]' (the first solution) to c0 and deletes itself.



P5 sends the message 'ap([a],[],[a])' to p0 and deletes itself. p0 produces p6 and deletes itself.



P6 sends message '[[a],[]]' (the second solution) to c0 and , finally, deletes itself.

3.2 Eager Interpreter Implementation

3.2.1 Eager Interpreter in Concurrent Prolog

Concurrent Prolog adopts AND-parallelism to describe concurrent processes and OR-parallelism to describe nondeterministic actions of processes (don't-care-nondeterminism). In Concurrent Prolog, once a clause is selected, the choice of other clauses is ignored. Concurrent Prolog provides interprocess communication mechanism (shared-variable) and process synchronization mechanism (read-only-annotation).

With the computational model implemented in Concurrent Prolog, a process is expressed by the following term:

```
process(Status,OutputChannel,
        InputChannel,Clause)
```

A generation of a process is performed by parallel AND's such as 'process:-process1,process2', and a deletion of a process is expressed by termination of the process, 'process:-true'. A channel is implemented by shared variables and process synchronization is achieved with read-only annotation. Although not shown in this paper, our system constructs the HDB using a meta representation, 'ax(Horn clause)', in the internal database of Concurrent Prolog. Fig.1 shows the program of the eager interpreter.

(p1) to (p3) define the behavior of active processes, while (p4) and (p5) define that of a waiting process.

(p1) performs reduction. The predicate 'reduce' checks whether or not the first element of the subgoals in 'Cls' is defined in the HDB. When the first element is not found in HDB, the predicate 'reduce' fails. When the guard portion of (p1) succeeds, two predicates in the goal portion, 'process' and 'process_fork', are executed in parallel. 'process' is the original process in waiting mode, and '?' is attached to the variable 'IC'. 'process_fork' generates a new active process for each newly fetched clause. 'merge' predicate is used for constructing a channel between a parent process and its child processes. Note that this merger deletes itself, when one input channel is closed.

(p2) corresponds to a process in a termination mode. The predicate 'terminate' checks that 'Cls'

```
p1) process(active,OC,Cls) :- reduce(Cls,NxGl) |
          process(wait,OC,IC?,Cls),process_fork(IC,NxGl).
p2) process(active,[Mess],Cls) :- terminate(Cls,Mess) | true.
p3) process(active,[],Cls) :- otherwise | true.
p4) process(wait,OC,[Mess|C1],Cls) :- newclause(Cls,Mess,NewC) |
          process(wait,OC1,C1?,Cls),merge(OC1?,OC2?,OC),
          process(active,OC2,NewC).
p5) process(wait,[],[],Cls).
    process_fork(OC,Goal) :- clauses(Goal,Clses) | forks(Clses,OC).
    forks([],[]).
    forks([Clause|R],OC) :-
          process(active,OC1,_,Clause),
          merge(OC1?,OC2?,OC),forks(R,OC2).
```

Fig.1 Eager Interpreter Program

is in the format 'X<--true'. The second argument '[Mess]' specifies that the message is sent to 'OC' and the active process is terminated. Then, the process deletes itself.

(p3) shows the operation of active processes in which further reduction has become impossible. (p3) deletes itself closing the output channel.

In (p4), the Input-Channel is a read-only variable; when a value is instantiated to the variable (i.e, when a message is received), the process starts operating. The predicate 'newclause' generates a copy 'NewC' from the original clause 'Cls' according to the waiting process operation definition mentioned in 3.1.2. The goal portion of the program specifies a new process generation with the new clause and the original process to remain as it was. The output channels of these two process ('OC1' and 'OC2') are merged into the original output channel 'OC'.

(p5) is for a waiting process with a closed message stream, which means that all the child processes have completed their jobs. The waiting process deletes itself closing its output channel.

Using this interpreter, the eager enumeration can be constructed as follows:

```
eager_enumerate([X|Y],Str) :-
   process(active,Str,(X<--Y)).
```

As described above, a computational model can be written in Concurrent Prolog very easily, because of its high descriptive capability. This also shows that OR-parallelism can be implemented by AND-parallelism.

3.2.2 The Refined Version of The Eager Interpreter

The eager interpreter described above is the direct implementation of the computational model in section 3.1. This implementation utilizes a 'merge' network for message communication. The merge predicate merges two streams nondeterministically to provide a characteristic of a channel where every child process can send a message to its parent independent of other child processes. However, the merge network has two drawbacks: it consumes a certain amount of the resources since a 'merge' is also a Concurrent Prolog process, and the message transfer takes relatively much time because the message is sent via more than one merger. By eliminating the nondeterminacy of the message transfer, we can construct a more efficient eager interpreter without the merge network.

The basic idea of the new version is to use D-list and linearize the channel. In this version, an input channel of a parent process is the concatenation of the output channels of its child processes. To achieve this feature, a reduction of an active process is changed as follows:

```
pr(active,OCs,OCe,(m<--a)) :-
 pr(wait,OCs,OCe,IC1?,IC,(m<--a)),
    pr(active,IC1,IC2,(a<--b1)),
    pr(active,IC2,IC3,(a<--b2)),
                  :
    pr(active,ICn,IC, (a<--bn)).
```

The first goal of the above clause specifies the parent process and the rest specifies its child processes. Each active process has both the output channel of its own (second argument) and its successor's output channel (third argument). After this

clause is selected, each child process computes its solutions to attach them to its output channel. Fig.2 shows the situation that the child_process1 produced two solutions, child_process2 produced one and the parent_process has received one solution 'sol11'. When a child process puts all the solutions into its output channel, it concatenates its output channel and its successor's channel. The parent process receives messages and executes its operation until the head pointer reaches the tail pointer of its input channel. When it terminates, a parent process concatenates its output channel and that of its successor because the parent process is a child process of the grandparent process. This method guarantees the ordering of solutions as well as OR-parallel execution.

4. LAZY ENUMERATION

This section introduces the lazy interpreter, which is a modification of the interpreter described in section 3. This interpreter provides the lazy enumerate function.

The lazy interpreter produces a solution for a given goal sequence according to the demand from one of the other Concurrent Prolog processes. Then the interpreter suspends the computation until it receives the next demand. When the interpreter receives a "kill" demand, it should release the resources and terminate itself regardless of its computational state. To implement the demand driven mechanism, the way of demand transfer and execution suspension control should be established. These are achieved by bounded buffer communication method in Concurrent Prolog (Takeuchi and Furukawa 83).

4.1 Bounded Buffer Communication

The interprocess communication is provided by the shared variables in Concurrent Prolog. Sending a message is instantiating a shared variable to the message. Since one instantiation corresponds to one message transfer, a new shared variable must be generated to continue the communication. According to Takeuchi, unbounded and bounded buffer communications can be supported in Concurrent Prolog.

The bounded buffer communication is achieved when the message receiver generates new shared variables. The following is a simple example of the bounded buffer communication with buffer length 2.

```
Goal :: integers(0,[X,Y|N?]),
        outstream([X,Y|N]\N)

integers(X,[X|M]) :-
        Y := X+1 | integers(Y,M).
outstream([X|M]\[P|R?]) :-
        wait(X)&write(X) |
                outstream(M\R).
```

'Integers' generates an integer

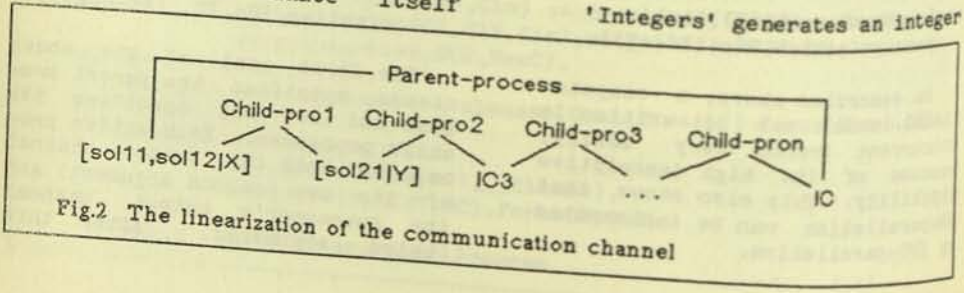


Fig.2 The linearization of the communication channel

stream. 'Outstream' outputs the elements of the stream. The symbol '\' is an infix operator which is used to write a head and a tail of D-list in one term. The call of 'integers' contains variables 'X,Y' which specify a buffer length of two. Process 'integers' can instantiate 'X' and 'Y' to 0 and 1 respectively, but cannot bind 2 to the variable 'N' because of its read only annotation. This process waits until the variable 'N' is bound. On the other hand, process 'outstream' waits until the 'integers' process binds the value because of the predicate 'wait(X)'. When the variable is bound to 0, 'outstream' writes the value and enters the recursive call. At this moment, a new variable 'P' is attached to the end of the communication channel because the tail of the channel (variable) is bound to '[P|R?]' in the head of 'outstream' definition. This instantiation enables the 'integers' process to continue the processing.

The bounded buffer technique enables the receiver process to control the sender process. Attaching an uninstantiated variable to the tail of the communication channel corresponds to the demand transfer from a receiver process to a sender process. Lazy enumerator communicates with other Concurrent Prolog processes via a bounded buffer as follows:

```
Goal lazy_enumerate({X|Y},[U|V?]),
     receiver([U|V]\V)
```

A 'kill' demand to an enumerator is to close the communication channel by binding '[]' to the tail of a channel.

4.2 Lazy Interpreter Implementation

Lazy Pure Prolog interpreter is obtained by changing the characteristics of the eager one as follows:

1) Replacing each communication channel from an unbounded buffer to a bounded buffer.
2) Using a linearized channel in-

```
p1) process(active,[],[],Cls).
p2) process(active,OPs,OPe,Cls) :- reduce(Cls,NxGl) |
            process(wait,OPs,OPe,[B|N]\N,Cls),
            process_fork(NxGl,[B|N?]).
p3) process(active,[Mess|R],R,Cls) :- terminate(Cls,Mess) | true.
p4) process(active,OPs,OPs,Cls) :- otherwise | true.
p5) process(wait,[],[],[]\_,_).
p6) process(wait,OPs,OPs,[M|_]\_,Cls) :- wait(M) & M='$end$' | true.
p7) process(wait,OPs,OPe1,[Mess|C1]\R,Cls) :-
      wait(Mess) & newclause(Cls,Mess,NewC) |
            process(active,OPs,OPe,NewC) &
            transfer_demand(OPe?,R,S) &
            process(wait,(OPe?),OPe1,C1\S,Cls).
    process_fork(Goal,OPs) :- clauses(Goal,Clses) | forks(Clses,OPs).
f1) forks(_,[]).
f2) forks([],['$end$'|_]).
f3) forks([Cls|R],OPs) :- wait(OPs) |
        process(active,OPs,OPe,Cls) & forks(R,OPe?).
    transfer_demand([_|_],[P|S?],S).
    transfer_demand([],_,[]).
```

Fig.3 Lazy Interpreter Program

stead of a merge network.
3) Adding process operations for a kill demand.

Fig.3 shows the program of the lazy interpreter.

(p1) to (p4) define the behavior of active processes. The second argument of an active process is its output channel and the third argument is its successor's output channel which is needed for linearizing a channel as mentioned in 3.2.2. When an active process is generated, its output channel is bound to '[B|N?]' or '[]'.

(p1) is a definition for manipulation of a kill demand, which specifies the termination of an active process. (p2) to (p4) correspond to the definitions in the eager interpreter. (p2) specifies the operation in the reduce mode where new child processes are generated and the active process changes its status to 'waiting' binding '[B|R?]' to its input channel. This binding is a demand for its child process. Fig.4 shows a demand transfer from a parent process to its child process. Predicate 'process_fork' executes 'clauses' and calls 'forks' which is to generate child processes. This process generation is controlled by the bounded buffer mechanism (the second argument of 'forks'). The second generation of a child process is postponed until the next demand is detected. (f1) specifies the behavior of 'forks' when a demand is to kill one. The serial-AND in (f3) specifies that a recursive 'forks' call should be tried after one process terminates. This is for only the efficient implementation in DEC-20 Concurrent Prolog which doesn't have a non-busy-wait mechanism.

(p3) and (p4) define that an active process terminates concatenating its output channel and its successor's (a unification of the second argument and the third one).

(p5) to (p7) defines the operation of waiting processes. (p5) which specifies a process termination is for a kill demand. When the message sent via its input channel is 'end', a waiting process concatenates its output channel and its successor's and terminates itself. The message 'end' means that all child processes of a waiting process are terminated. (p7) specifies a waiting process operation when it has received a solution. Fig.5 shows the configuration of an output channel of a waiting process and that of a new active process. Output channel 'OPe' will be attached to the tail of the output channel of 'new process' when it terminates. Predicate 'transfer_demand' in (p7) transfers a

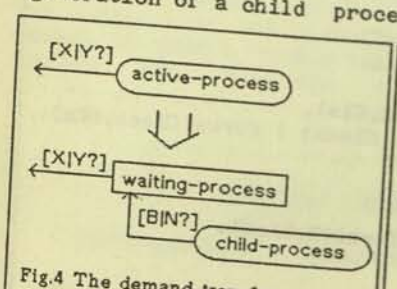


Fig.4 The demand transfer

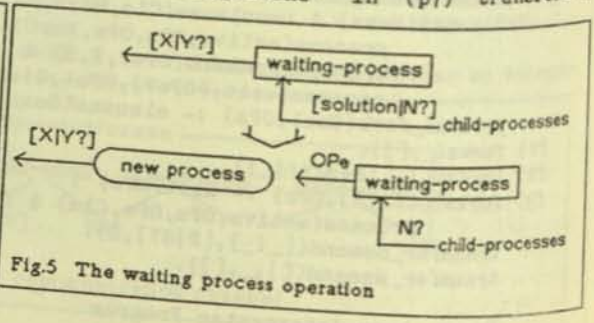


Fig.5 The waiting process operation

demand, for example, the waiting process in the above figure instantiates 'N' to '[B'|N'?]' or '[]' according to a demand it receives.

Using the lazy interpreter, 'lazy_enumerate' is defined as follows:

```
lazy_enumerate({X|Goals},OPs) :-
        process(active,OPs,OPe,
                (X<--Goals)) &
        sendend(OPe?).

sendend([end_of_solution|_]).
sendend([]).
```

'Sendend' sends message 'end_of_solution' when a demand number exceeds the total number of the solutions. The demand-sender process receives 'end_of_solution' instead of a solution when it has received all solutions.

The interface between 'lazy_enumerate' and other Concurrent Prolog process is a bounded buffer.

5. DISCUSSION

To realize don't-know-nondeterminism, an environment of variable bindings must be maintained for multiple solutions. The interpreter described in this paper retains the environment by copying a clause, that is, a waiting processes copies its clause when it receives a message. A simple copying method has drawbacks on both space and time efficiencies.

The space problem is that a simple method produces a whole copy of a given term which contains non-variable portions which can be shared. This problem is avoided by introducing a 'rename' predicate which produces a copy of a term sharing ground term portions with its original term.

The time problem is that a copy operation should search the whole part of a given term. This will increase a computation time of a waiting process according to the size of the terms it contains. One of the possible optimization methods for this problem is to determine the portion to be shared in compile time (either automatically or by giving declarations). The development of an efficient renaming method is one of the important topics for implementation of the don't-know-nondeterminism.

6. CONCLUSION

This paper described an OR-parallel execution model for Pure Prolog and the implementation of an enumerate function in Concurrent Prolog based on the model.

The computational model is based on multi-processing and interprocess communications. The model provides an eager Pure Prolog interpreter implemented in Concurrent Prolog. Also a lazy interpreter can be obtained easily by introducing a bounded buffer communication mechanism to the eager interpreter. The eager interpreter and the lazy interpreter provides eager and lazy enumerate functions to Concurrent Prolog, which are very important functions for parallel logic programming.

This approach shows that both OR-parallel and AND-parallel execution of a logic program is achieved only by AND-parallel execution. This feature is very important because it decreases the complexity of the computer architecture and the amount of required hardware of a highly parallel machine.

ACKNOWLEDGEMENT

We would like to thank Mr. Takeuchi for his valuable suggestions on the use of Concurrent Prolog and on the computational model. We would also like to thank Dr. E. Shapiro and Dr. K. Kahn for their useful advice about the OR-parallel execution of logic programs.

REFERENCES

Clark,K.L and Gregory,S PARLOG: A Parallel Logic Programming Language. Imperial College Research Report, (May 1983).

Conery,J,S The AND/OR Process Model for Parallel Interpretation of Logic Programs. Technical Report 204, University of California Irvine, (1983).

Emden,M.H. and de Lucena Filho, G.J. Predicate Logic as a Language for Parallel Programming. in LOGIC PROGRAMMING. Clark,K.L. and Tarnlund,S.A. eds., Academic Press, (1982).

Furukawa,K. and the Kernel Language Design Group. Conceptual Specification of the Fifth Generation Kernel Language Version1 (KL1). to appear as an ICOT Technical Report, (1984).

Haridi,S. and Sahlin,D. Evaluation of Logic Programs based on Natural Deduction. The Royal Institute of Technology, TRITA-CS-8305, (1983).

Hirakawa,H., Onai,R. and Furukawa,K. Implementing an OR-Parallel Optimizing Prolog System (POPS) in Concurrent Prolog. ICOT Technical Report, TR-020, (1983).

Kahn,K.M. A Primitive for the Control of Logic Programs. UPMAIL Technical Report no. 16, Uppsala University, (1984)

Kowalski,R. Logic for Problem Solving. North Holland, New York (1979).

Nitta,K. Matsumoto,Y. and Furukawa,K. Prolog Interpreter Based on Concurrent Programming. Proc. of 1st International Logic Programming Conference, pp.38-42, (1982).

Shapiro,E.Y. A Subset of Concurrent Prolog and Its Interpreter. ICOT Technical Report TR-003, (1983).

Shapiro,E.Y. The Bagel: A Systolic Concurrent Prolog Machine. to appear as an ICOT Technical Report, (1984).

Takeuchi,A and Furukawa Interprocess Communication in Concurrent Prolog. Proc. of Logic Programming Workshop, (1983).

Warren,D.H. Higher-Order Extensions to Prolog: Are They Needed? D.A.I. Research Paper No. 154, (1981).

Incorporating Mutable Arrays into Logic Programming

Lars-Henrik Eriksson and Manny Rayner

Uppsala Programming Methodology and Artificial Intelligence Laboratory
Computing Science Department, Uppsala University
P.O. Box 2059, S–750 02 UPPSALA, Sweden

This work was supported by the National Swedish Board for Technical Development (STU).

1. Abstract

Logical terms are the only compound data structures in logic programming languages such as Prolog. Terms are sufficiently general that no other data structures are needed. Restricted uses of terms correspond to the bits, character strings, arrays, records, etc. of other programming languages. The computational overhead, however, of using a very general data structure in specialized situations can be very high. Side-effects cannot be performed upon logical terms and the alternative of constructing new terms which differ slightly from the old can be very costly. We propose to alleviate these short-comings of terms, without losing their logical clarity and purity.

We have introduced into LM-Prolog, a Prolog dialect running upon Lisp Machines, predicates for creating and manipulating arrays. These predicates could have been written completely as Horn clauses without the use of any primitives. They are implemented in terms of physical arrays and "virtual arrays" in a manner that is transparent to the user. For some uses of these predicates, it is possible for a compiler to produce code performing array references and updates that is as good as that produced by compilers for traditional programming languages.

2. Motivation

The goal of this research is to significantly improve the efficiency of some logic programs without sacrificing their logical purity. First we will consider where the use of arrays can improve performance and then address the question of whether a complex implementation just to maintain logical purity is worthwhile.

There is a growing interest in attempting to extend the domain of logic programming to systems programming. The Japanese fifth generation project exemplifies this [Chikayama 1983]. It is difficult to imagine an efficient file system or editor which does not do side-effects upon compound structures. We believe that logical arrays as described in this paper provide a viable alternative to non-logical primitives which perform such side-effects.

There are many existing Prolog programs whose performance could be enhanced by using logical arrays. A chess or go program can represent the board as a two-dimensional array. A program using association lists could instead use hash tables. A text processing program that deals with text as lists of character codes can be replaced by one using character arrays. And so on.

One may question our insistence that arrays be incorporated into logic programming in a *logical* fashion. Prolog already has non-logical predicates for i/o and changing the database why not add a few more for dealing with arrays? Admittedly the implementation of such predicates would be simpler than the one we describe later. The reasons why we want the implementation to remain within logic are both theoretical and pragmatic. If we can maintain a simple semantics for extensions to Prolog then the job of verifying, synthesizing, analyzing, optimizing, debugging or understanding logic programs will be easier. Pragmatically, by introducing arrays into Prolog in a logical fashion we maintain the generality of the logic programs. A Prolog predicate that, for example, works for any instantiation pattern and that uses lists can often be made more effective by using arrays without any loss of generality. Logical arrays show promise of being well-suited for implementations upon parallel processors by avoiding the usual synchronization problems associated with concurrent programs with side-effects.

3. Semantics

This chapter is intended to describe the semantics of our array primitives using pure Prolog procedures with arrays represented as lists.

3.1 Array processing primitives

We define three primitive predicates to deal with arrays:

is_array(Array,N)

Array is an array with **N** elements. This predicate is used both for creation and type checking, depending on the instantiation of the arguments.

array_element(Array,Index,Value)

The value of the element corresponding to the index **Index** of the array **Array** is **Value**. This is the primitive for accessing array elements.

array_update(Old_array,Index, Value,New_array)

The array **New_array** is the same as array **Old_array**, except that the value of the element corresponding to the index **Index** in **New_array** is **Value**. This is the primitive for updating arrays.

3.2 Horn Clause Definitions

In the sequel, we will assume that the semantics of the array processing primitives are as if they were defined by the following Horn clauses. **array** is a reserved functor symbol that cannot be used anywhere else.

```
is_array(array(Array_list),Size) :-
  length(Array_list,Size).

array_element(array(Array_list),
              Index,Val) :-
  nth_element(Array_list,Index,Val).

array_update(array(Old),Index,
             Val,array(New))) :-
  update_element(Old,Index,Val,New).

/* Auxiliary predicates */
length([],0).
length([_|List],Len) :-
  Len>0, Len1 is Len-1,
  length(List,Len1).

nth_element([Elem|_],0,Elem).
nth_element([_|List],Pos,Elem) :-
  Pos>0, Pos1 is Pos-1,
  nth_element(List,Pos1,Elem).

update_element([_|List],0,New_elem,
               [New_elem|List]).
```

```
update_element([Some_elem|Old_list],
               Pos,New_elem,
               [Some_elem|New_list]) :-
  Pos>0, Pos1 is Pos-1,
  update_element(Old_list,Pos1,
                 New_elem,New_list).
```

4. Implementation

The Horn clauses describing the semantics of the array predicates can be executed by a Prolog system. There are, however, two sources of inefficiency. The use of lists precludes any random accessing. This is not a serious problem since terms can be used as described in section 5.2. The serious problem is the copying that **update_element** does. Our implementation avoids both of these problems by using arrays. We manage to do this in a way that the semantics of the predicates is not violated. An array used by these predicates is implemented as a real array and a data structure which masks the values of some of the array elements. Both the old and new versions of an array being updated share the same real array.

This chapter will describe the principles of the implementation of mutable arrays and the primitive operations on them. The implementation requires that certain arguments to the primitives are instantiated when the primitives are called. If they are not, our implementation cannot handle the case in other ways than signalling an error, freezing [Colmerauer 1982] the subgoal until enough arguments are instantiated, or possibly successively binding (by backtracking) an index argument to all possible indices.

Mutable arrays are represented internally by a chain of *value blocks*, terminated with an physical array. In the rest of this chapter, the term "array" will refer to the entire mutable array. When the underlying physical array is referred to, the term *real array* will be used. There should not be any way for a logic program to look at the internal structure of a mutable array.

The terms *old* and *new* (or *updated*) array will be used to refer to the mutable array to be updated and the mutable array created as a result of the update (in a procedural sense). Note that even though the values of the elements stored in the old array are unchanged by an update (since pure logic programming is side-effect free), the *structure* of the old array might change. Thus we will refer to the old array *before* and *after* the update. Since this chapter describes the actual implementation of mutable arrays, this procedural view of Prolog will be used throughout.

Each value block - except the last one in a chain - contains an *index-value pair*. The value of a certain element of the mutable array is the value stored in the real array, *unless* one of the value blocks in the chain has that index. In the latter case, the value of the element in question is the value stored in the first value block with that index instead. Two mutable arrays which differ only in the values of a few elements, sometimes share the real array, the differences being handled by the value block chains.

Aside from the index and value, a value block contains a *pointer* to the next value block in the chain, or to the real array, should this be the last value block of the chain. If the value block is the last one in the chain, it will not actually contain a valid index-value pair. This can be represented by a flag bit, by some special code in the index or

value fields or simply by the fact that the value block points to a real array.

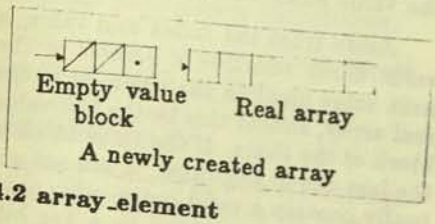
The *new-real* scheme update algorithm (see below) requires that the real array is never pointed to by more than one value block at a time. The sole purpose of this last value block is to fulfill this requirement and still provide for different mutable arrays to have different value block chains pointing to the real array. All the chains simply share the same last block.

4.1 is_array

The predicate **is_array** requires that at least one of its arguments is instantiated.

When **is_array** is called with the **Array** argument instantiated, its dimension is unified with the dimensionality argument.

When the call is made with the **Array** argument uninstantiated, a mutable array will be created. To create the mutable array, a real array is allocated for it. Each element of the real array is initialized with unbound variables (or possibly some given term). A single value-index block is allocated, pointing to the real array. The value-index block is flagged as not containing a valid value-index pair. A reference to the value-index block is returned as the reference to the new mutable array. The real array is never pointed to directly.



A newly created array

4.2 array_element

The **Array** and **Index** arguments must be instantiated. Looking up the value of an element of the mutable array is done by comparing the index of the element to be looked up to the index in each value block in turn (except for the final value block which does not contain a valid index-value pair). Should a block with a matching index be found, the corresponding value is the one looked for. If no such value block is found, the real array is indexed in the ordinary manner. When the value is found, it is unified with the **Value** argument to **array_element**.

4.3 array_update

All arguments to **array_update**, except possibly the **New_array** argument, should be instantiated. If the **New_array** is instantiated, but the **Old_array** argument is not, it is still possible to handle the call by using the fact that

```
array_update(Oldarray,Index,
                Value,Newarray)
```

can be replaced by

```
array_element(Newarray,Index,Value),
array_update(Newarray,Index,_,Oldarray)
```

which switches the old and new arrays.

When an element of a mutable array is updated, a copy of the array is actually made to preserve the value of the old array. This copying is done in such a way that the real array is always shared with the old array. In addition, the internal structure of the old array might be changed by the updating. This change is, of course, done in such a way that is does not alter the value of any of the elements of the old array.

Updating can be done using two different schemes, called the *old-real scheme* and the *new-real scheme*. In the old-real scheme, the value blocks

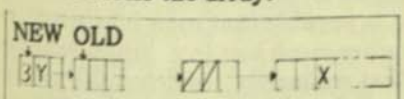
are used to keep a history of all changes made to the array. In the new-real scheme, the real array is actually changed and the value blocks are used to keep a history of the *old* values before the updates, for the benefit of the old array.

In most cases the programmer can specify, as control information, which of the methods he wants to use (both schemes are semantically equivalent); however the new-real scheme fails in a few cases, in which cases the system should do an old-real update instead. In both cases, the update concludes by unifying the new array with the **New_array** argument to **array_update**.

The names refer to whether the old or the new array will consist of only the real array and the compulsory value block without a value, when a freshly created array is updated.

The old-real scheme

This scheme is the simplest one. The new array is simply a new value block, containing the index and value of the updated element, pointing to the old array. In this scheme the structure of the old array is not changed at all, but the new array has an index block chain one element longer. If the old-real scheme is used several times in succession, a long chain of value blocks will be built up, increasing the time needed to access the array.



The situation after the third element of the old array has been set to Y (previous value X), using the old-real scheme.

The new-real scheme

The idea of this scheme is to update the real array and add a value block containing the previous value to the value block chain of the old array to mask the change. In this way, access to the new array will be as efficient as access to the old array was before the update took place. Instead, access to the old array will be slowed down by the addition of a value block.

An update using the new-real scheme takes place in a number of steps:

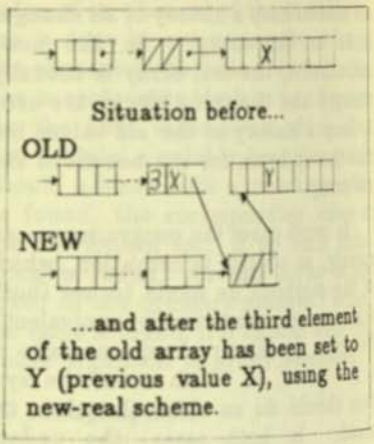
1. The new array is created by copying the value block chain of the old array, omitting any value block whose index is the same as the index of the element being updated.
2. The index of the element being updated together with its current value in the real array is stored into the (hitherto unused) last block in the old value block chain.
3. The pointer in that value block is set to point to the last value block in the copied chain, rather than to the real array.
4. The new value of the element to be updated is stored into the real array.

Note that the values of all elements of the old array are unchanged by this operation, since the storing of the old value in step (2) masks the update of the real array in step (4). We know that all mutable arrays using this real array will point to the real array through the value block used in step (2), so the masking applies equally well to other arrays than the old array of this particular update. The new array, on the

other hand, is identical to the old array, except for the updated element, since that element was changed in the real array in step (4) and any value blocks that could have masked the changed was omitted when the chain was copied in step (1). The reason this copy was necessary was both to omit any blocks that could have masked the change as well as preventing the new value block from masking the change from the new mutable array.

The motivation for step (3) is to ensure that there is still only one value block that points to the real array, so that step (2) in later updates will work. Step (3) also increases the length of the old value chain by one value block. When the old array is not referred to any more (which probably happens soon after the update), its value block chain could be garbage collected. Since the new value block chain is not longer (but possibly shorter) than the old chain, programs that don't use old arrays after they have been updated will run using constant space.

This procedure can be simplified if the value block chain contains a block for the element to be updated, already containing the new value, or if the real array element already contains the new value. (Note that the old value could still be different.) In the former case, the copying in step (1) is only done up to, but not including the value block in question. In the latter case the copying is done up to but not including the last value block if the real array contained the desired value. The last block in the new chain is left pointing to the place in the old chain where the copying stopped, and steps 2-4 are skipped.



In order to undo the side-effects performed on the old array in steps 2-4 on backtracking, the pointer fields and real array elements need to be trailed [Warren 1977]. Prolog trailing commonly only involves just recording the address of the location that was set, as only unbound variables can be changed in Prolog. To reset these variables to an unbound status, only their address need be known. This simple trailing is not sufficient here. Rather, the previous contents of the changed cell must be trailed along with the address of the cell. This requires that the trail has space to store the old contents, or that two trails, one with and one without previous values, are used.

If the elements of a real array have fewer bits than are normally required to store information in the particular Prolog system (e.g. a character array, where all elements are eight bits wide), it is possible that some information requires more bits (e.g. logical variables, that ordinarily requires a full pointer) and thus cannot be stored in the real array in step (4). If an attempt is made to update an array element with that information, the new-real scheme cannot be used, since it always updates the

real array. The old-real scheme must be used instead.

Comparison

In the old-real scheme the value block chain of the updated array will be one block longer that that of the old array, while in the new-real scheme it will have the same length. This means that if several updates in succession are done on successive versions of an array, the size of the value block chain (and thus the time needed to access the array) will be constant for the new-real scheme, but linearly increasing for the old new scheme. On the other hand, the time to access the earliest version of the array will increase linearly as new versions get updated.

In most cases, the updated array is probably going to be used more than the old array, so the new-real scheme will usually be advantageous. If the old array will continue to be used extensively, the old-real scheme might be more efficient as the old array is not affected in any way by the updating.

Another factor to the disadvantage of the new-real scheme, is that it has to copy the value block chain of the old array. Should this chain be long, the copying could take substantial time and memory.

4.4 Unifying arrays

The unification mechanism of Prolog must be extended to deal with arrays. In accordance with our Horn clause definition, we define two arrays to be unifiable iff they have the same dimensionality and corresponding elements unify. The occur check is as much (or little) applicable here as when standard terms are unified. Since the Horn clause definitions use a reserved functor to represent arrays, an array can never unify with something that is not another array.

4.5 Realization in LM-Prolog

The LM-Prolog implementation of the array predicates is generalized somewhat. The **index** and **dimension** are replaced by **indices** and **dimensions** which are lists which can have up to seven elements. This extension could clearly be written in pure Horn clauses. **Is_array** takes an extra argument which is a list of options. The options can be used to declare whether the elements of the underlying real array should be full words, 16 bits, 8 bits, 4 bits, 2 bits, or single bits. This option has no affect upon the semantics of the primitives and can be viewed as user-provided control information. Another option specifies how, if at all, the array should be initialized. This too could easily be written in pure Prolog.

The development and implementation of logical arrays and experimentation with various optimization was greatly facilitated by the extent to which LM-Prolog is designed to be extensible by users ([Kahn 1984] and [Carlsson 1983]). The entire array package was written in Lisp and Prolog without making any changes to the underlying implementation of LM-Prolog. Unification in LM-Prolog is extensible by using the Lisp Machine's Flavor message passing facility [Moon 1983]. Logical arrays are implemented as flavor instances that receive messages to unify with others, to lookup elements, to perform updates, to copy themselves, etc. Another important facility is the ability to "trail" arbitrary computation. In the normal execution of Prolog programs a trail of cells which upon backtracking need to be re-set to unbound is kept. In LM-Prolog, the trail consists of both such cells and Lisp forms to be executed. In order to implement backtracking

when the underlying implementation performs side-effects upon arrays, it was necessary to trail array locations and their previous values. Another factor which facilitated the implementation of logical arrays in LM-Prolog is the smooth interface to the facilities of the underlying Lisp system. Lisp Machine Lisp has an excellent array facility which supports various byte sizes, array dimensions, and overlaying.

5. Optimizations

Programs using logical arrays are often substantially more efficient than programs built upon the existing alternatives in logic programming languages. If, however, logic programming is to compete with Lisp or Pascal then we must consider carefully the overhead involved in supporting logical arrays. In general, the overhead is necessary, but there are commonly occurring uses of logical arrays that could be significantly optimized.

Our attitude towards optimizations of logic programs is summarized below. It is based upon a distinction between control advice that cannot change the logic of a program and declarations that state that a program fulfills certain properties. If a declaration is incorrect then the execution of the program may also be. In descending order of desirability we classify optimizations as follows:

- Automatically detected and performed optimizations. E.g. tail recursion optimizations.
- Control advice. E.g. that the underlying array be a byte array.
- User declarations. E.g. the mode declarations of Dec 10 Prolog and the array usage declarations described below. Ideally these declarations should be verified either by the system or the user. Failing that the system should optionally do run-time checking.

There are three kinds of control advice one can give the LM-Prolog implementation of arrays. One is what the byte size of the underlying real array should be. If, for example, one declares the size to be a single bit, then the array can efficiently hold only 0 or 1. Any other value, including an unbound variable, is captured in a value block which costs a few words of memory and slows down array accesses. If, however, the vast majority of values in an array are limited to 0 or 1, LM-Prolog can pack 32 values into each machine word.

Another type of control advice is when an array should be copied. If an array with a long chain of value blocks is used frequently, either directly or *via its ancestors*, then it may be worthwhile re-representing it as a real array. This real array's contents are the same as those of the old logical array. This operation takes time and memory but can be critical for sufficiently fast element access. Logically, this copying is just advice to the system and does not change the semantics of the programs involved.

One can consider the choice between old-real and new-real updates as control advice. If older versions are used more frequently than newer ones then one should advise the system to use old-real updates.

The usage declarations that LM-Prolog accepts describe whether the array will be used in a deterministic manner, whether only the most recent version of an array will be accessed, whether the array indices in a lookup or update will always be ground, and if a byte array is being used that all

of its values will fit. If an array's usage is declared to be deterministic from its creation to its last update, then no trailing is performed. The idea is that if the system backtracks to the last update, then it will backtrack all the way back to the creation of the array, so it does not matter if its elements are inconsistent.

It is quite common to use an array in a linear fashion so that after an array is updated, the old version is never used. This is the way arrays are used in traditional programming languages. If an array is declared to be used in this fashion then an update can simply perform side-effects upon the real array of the old array reference since the old array won't be used anymore.

The logical implementation of arrays must be prepared to find that the array indices in a lookup or update are only partially instantiated. The cost of checking first if they are ground is small but can be optimized away if the array is declared to be used only with ground indices. Similarly, byte arrays must check that the value is an integer in the proper range and this check can be declared away.

If the usage of an array is declared as deterministic, recent version only, ground indices and proper values for byte arrays then the efficiency of the current implementation is about one-half of that of Lisp. This is because, in addition to the array reference or update, a message must be sent. With a little micro-code support for arrays this overhead could become insignificant. A micro-coded primitive could if passed a real array do the ordinary array reference or update, otherwise send a message to the flavor instance.

In order to verify a logic program containing array usage declarations, the declarations must be shown to be correct. The verification of usage declarations is an important area for future research. The implementation does allow one to declare that a declaration be checked at run-time and that an error be signaled if it is violated.

6. Element-wise Alternative

Under certain circumstances, the implementations discussed above may become inefficient. Suppose that the array is large; that is to say, that it will be a very expensive operation to copy it. Suppose moreover that a large number of updates will be carried out on it, and that it will be necessary to access not only the most recent, but also old states. Such a situation could easily arise, for example, if the array is being used to represent the state of a large dynamic system, and the object is to collect states which fulfill some condition, in order to compare them later. The problem here is that every array access will have to "go through" all updates between the version in question and the most recent version; however, in general each individual element will only have been updated a small number in times compared to the total number. It is thus apparent that a very substantial optimization can be performed if we "localize" update information to the element affected. How this could be done is now discussed.

The basic scheme is similar to that used by Conniver [McDermott 1974]; an array is a triple consisting of a *real-array*, a *version-identifier tree* and a *version-identifier*. The version-identifier is unique for each version of an array, and consists of a list of integers. The version-identifier tree is

a tree which contains all the version-identifiers current for the array in question, partially ordered by the relation *older than*. One can reasonably think of the version-identifier as a "Dewey decimal" number; then the version-identifier tree is a catalog of all the versions of the array that have been created.

The relationship *older-than* we define as follows: if v_1 and v_2 are two version identifiers, then v_1 is older than v_2 iff either

1) the list v_1 is an initial segment of the list v_2, or

2) the two lists are identical except for the last elements, and the last element of v_1 is less than the last element of v_2.

Each element in the real array contains a list of pairs, the *local a-list*, each pair containing a value and a version identifier. To find the value of an element for an array with real array **a** and version identifier v we go down the a-list until we find the value paired with the first version identifier that is older than or equal to v. Conversely, to update the array we first generate a new version identifier v' in the following way: if it does not already exist in **a**, we form the identifier that is the same list as v except for the last element, which is incremented by one; otherwise, it is the list with a 1 appended to the end. v' is then inserted into the appropriate place in the version-identifier tree, and a pair composed of v' and the new value is inserted into the local a-list for the index in question immediately after the first version-identifier older than v.

The efficiency of this scheme is hard to estimate, since it depends on several independent variables. The most important of these is the way in which the arrays are updated; if this is done "linearly", so that the version-identifier tree only has a single branch, the version identifiers are all single-element lists and both updating and referencing are fairly efficient. In general, the "bushier" the version-identifier tree the worse the method will perform, since the lists representing the version identifiers will become longer and the overhead in processing them correspondingly greater. Another important question is how evenly the updates are spread through the array; clearly, if they are concentrated on a small proportion of the elements the method is correspondingly worse. One definite disadvantage of this scheme compared to those above is that garbage collection would be very expensive, but since it is only intended to be used in cases where old values are of interest this is perhaps not serious. Also, it is impossible to implement byte-arrays, since each element in the real array must be able to hold an arbitrary value.

7. Dec 10 Prolog Alternative

Many Prolog implementations have the predicates **functor** and **arg** which can be used to define efficient versions of the array predicates ([Pereira 1979] [Clocksin 1981]). Prolog terms can be used as *pure* arrays using these primitives.

The difficulty with this approach is the lack of an efficient way to implement **array_update** under the new-real scheme. Recall that the cost of a lookup of an array updated under the old-real scheme is proportional to the number of times that array has been updated. Under the new-real scheme one needs to perform side-effects upon the real array. Since such side-effects

are not possible on terms, one is forced to consider other unattractive alternatives such as adding and removing clauses from the Prolog database to implement the side-effects.

8. Concurrent Prolog Alternative

Another way to obtain the advantages of mutable arrays and yet remain logical is to use the Concurrent Prolog technique of defining processes which accept messages [Shapiro 1983a]. Mutable arrays can be implemented this way by having each array element correspond to an argument to the "array" process. The process accepts message to "look up" values and to "update". A schema for writing such processes is given below.

```
array_n([lookup(1,Element_1)|MoreMsgs],
        Element_1,...,Element_n) :-
    array_n(MoreMsgs?,
            Element_1,...,Element_n).

...

array_n([update(1,NewValue)|MoreMsgs],
        Element_1,...,Element_n) :-
    array_n(MoreMsgs?,
            NewValue,...,Element_n).

...
```

The practicality of such an implementation of mutable arrays depends upon sophisticated argument passing and tail-recursion optimizations. An awkwardness is the need to define different processes for each array size. One way around this is to define one or a few standard size arrays, and build larger ones as arrays of arrays. The complexity of a lookup or update would then be $O(\log_s(n))$ where s is size of a standard array and n the size of the array being accessed. It can be easily seen that this implementation of arrays will not work in Prologs based upon depth-first search.

9. Arrays as Impure Predicates

An alternative way to provide arrays in Prolog is to implement each array as a separate predicate. Given the appropriate indexing advice, a Prolog system could use an array for indexing. Array updates would be realized by database updates. Such an implementation of arrays is similar to what traditional programming languages provide. Older versions of the array are not accessible, there is no backtracking, and programs become more sensitive to the order in which its parts are executed. The semantics of such arrays becomes the semantics of database updates.

10. Other Alternatives

Recently, two other schemes for representing arrays in Prolog have been suggested. The idea described in [Cohen 1984] is extremely ingenious, but appears to suffer from some serious problems; access time in the general case is proportional to the total number of updates, and garbage collection is very difficult. Also, if a virtual array is replaced by a concrete one, the change leaves versions logically dependant on the changed version unaffected. This makes optimization by "concretization" very inefficient.

[Pereira 1984] gives a representation of arrays as trees, with access in logarithmic time and update logarithmic in both time and space. One attractive feature is that it is possible to postpone allocating space to an entry until it is updated for the first time. This makes the idea very suitable for sparse arrays, but in the general case it is not

clear that the overhead is acceptable for a large array size.

11. Applications

The use of mutable arrays on conventional hardware can easily be defended in terms of efficiency. Even in the areas where Prolog is usually applied, symbolic processing and databases, arrays can be important. Hash tables, for example, implemented using arrays are often used in symbolic processing. While all serious Prolog implementations use hash tables, a Prolog user has no access to the hash table routines and is forced to use less efficient alternatives.

Graphics is another area where arrays seem appropriate. Typically changes are made to some small part of a model of a two or three dimensional space. Logical arrays provide the exciting possibility of exploring the idea of doing computer animation where the entire history of the display needs to be computed. Considering the millions of bits each frame needs and the more than a thousand frames needed each minute, one needs something like logical arrays to store the different versions.

Two applications are currently being implemented by the authors of this paper. A Go program based on the principles described in [Rayner forthcoming] represents the board with mutable arrays. This makes good use of the special features available, since one is normally interested in examining a large number of positions which only differ slightly from the current one. Another project with a more direct relevance to logic programming is a natural deduction-based proof system of the type described in [Haridi 1983]. A problem that arises is that it is necessary to keep several different binding environments simultaneously; this occurs in a large variety of "parallel" logic programming systems. Implementing environments as a-lists is clearly too expensive, but by regarding variables as offsets into an "environment array" it is possible to use the methods described here to provide an efficient solution inside pure logic. This will be discussed more fully in a later paper.

12. Discussion

On top of logical arrays one can, within LM-Prolog, provide strings, hash tables, general record structures and the like. There are two questions here that warrant more research. Is this a good way of providing such capabilities? Should such facilities be built on top of arrays or should they be provided in a manner analogous to how arrays are implemented?

The latter question is rather system dependent. Since the Lisp Machines provide well-designed hash-tables, record structures, and strings with significant micro-code support it would seem sensible to take advantage of them and implement the "logical" versions of them in Lisp rather than Prolog.

The question about whether this is the right way to introduce, say, strings into Prolog is less clear. A disadvantage of logical arrays is that they are awkward to compute in comparison to lists. Unification of lists provides a very succinct and clear way of expressing something that may require a series of calls to **array_element** and **array_update**. Perhaps strings should be introduced into logic programming as terms that one can perform string unification upon. Or perhaps they

should remain as lists of characters as they are in many Prolog implementations and effort should, instead, be put into packing several characters to a word [Shapiro 1983b]. Another alternative is to implement strings as logical arrays and put effort into extending unification to enable one to deal more comfortably with both strings and arrays.

We presented the "old-real" and "new-real" schemes which can co-exist side-by-side. We also discussed an alternative element-wise scheme which for some uses of arrays was ideal. An interesting avenue of future research is how to let the system choose the appropriate underlying representation depending upon how the arrays are used.

We have only begun to consider the design of generally useful utilities for manipulating arrays. We expect that the ability to perform some operation upon each element of an array, to create arrays that are pieces of other arrays, and the like to be desirable. APL [Iverson 1962], for example, is successful not because it provides array referencing and updating, but because it provides a rich and powerful set of tools built upon those primitives. One primitive that we are exploring is **array_differences** where depending upon how its used can perform parallel operations upon an array or find differences between two arrays.

One motivation for providing mutable arrays in a pure fashion is that the resulting techniques and algorithms apply equally well in the context of functional programming. Our introduction of virtual or logical arrays to logic programming applies equally well to functional programming. Virtual arrays may also be useful in Lisp and message passing systems. The LM-Prolog implementation is really in two layers. First, virtual arrays are implemented as actors (flavor instances) and then interfaced into LM-Prolog.

An interesting area for further research is to consider logical arrays in the context of parallel processing. Clearly the old-real update works well in the face of concurrency since there are no side-effects. The new-real implementation has side-effects that are completely hidden from the user. Could the problems of simultaneous updates also be handled by the implementation in a transparent manner?

We have began to work on manipulating Prolog databases in a manner analogous to mutable arrays. Both the old-real and new-real schemes have database analogs. The three array primitives are replaced by primitives to create, query, and modify databases. An awkwardness of this scheme is that one must explicitly provide a database argument to the calls of Prolog predicates using mutable databases. The advantages of maintaining a pure semantics typically outweigh this clumsiness. Perhaps a syntactic sugar for defaulting database arguments is feasible.

In summary, the introduction of logical arrays into Logic Programming is very promising. The range of programs that can be effectively run in logic has been expanded. The unique ability to use old versions of arrays supports many new applications. In the long run, experience with using logical arrays will decide how useful they really are.

13. Acknowledgements

We would like to acknowledge the contribution made to this work by Dr. Ken Kahn, who but for technical rea-

sons would have appeared as a co-author.

14. References

[Carlsson 1983] Carlsson, M., Kahn, K., "LM-Prolog User Manual", UPMAIL Technical Report No. 24, Uppsala University, Sweden, Nov. 1983

[Chikayama 1983] Chikayama, T., Yokota, M., Hattori, T., "ESP – Extended Self-contained Prolog as a Preliminary Kernel Language of Fifth Generation Computer", *New Generation Computing*, Vol. 1, No. 1, Tokyo, Japan, 1983

[Clocksin 1981] Clocksin, W. and Mellish, C., *Programming in Prolog*, Springer-Verlag, Berlin, Hiedleberg, New York 1981

[Cohen 1984] Cohen, S., draft, University of California, Berkeley

[Colmerauer 1982] Colmerauer, A., "PROLOG II Manuel de Reference et Modele Theorique", *Proc. Prolog Programming Environments Workshop*, Linkoping Sweden, March 1982

[Haridi 1983] Haridi, S., Logic Programming Based on a Natural Deduction System, Doctoral Thesis, Royal Institute of Technology, Stockholm Sweden, 1983

[Iverson 1962] Iverson, K., *A Programming Language*, Wiley, New York, 1962.

[Kahn 1984] Kahn, K., Carlsson, M., "How To Implement Prolog on a Lisp Machine", in *Issues In Prolog Implementations*, ed. Campbell, J. Ellis Horwood Ltd., West Sussex, Great Britain, 1984

[McDermott 1974] McDermott, D., Sussman, G., "The Conniver Reference Manual", MIT AI Laboratory Memo No. 259a, Cambridge, USA, January 1974

[Moon 1983] Moon, D., Stallman, R. M., Weinreb, D., "Lisp Machine Manual", MIT AI Laboratory, January 1983

[Pereira 1979] Pereira L., Byrd L., Pereira F., Warren D. H. D., "User's Guide to DECsystem-10 Prolog", DAI Occasional Paper 15, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland

[Pereira 1984] Pereria F., Arpanet Prolog Digest, Vol 2, No. 12, March 14 1984

[Rayner forthcoming] Rayner M., "The concepts of sente and aji in Go and chess", Upmail Technical Report, forthcoming

[Shapiro 1983a] Shapiro, E., *A Subset of Concurrent Prolog and Its Interpreter* ICOT Technical Report, TR-003, ICOT, Tokyo, 1983

[Shapiro 1983b] Shapiro, E. Informal presentation at ICOT, Tokyo, November 1983

[Warren 1977] Warren, D. "Implementing Prolog – compiling predicate logic programs", Department of Artificial Intelligence, University of Edinburgh, D.A.I. Research Report Nos. 39 and 40, May 1977

Equality, Types, Modules and Generics for Logic Programming[1]

Joseph A. Goguen and José Meseguer
SRI International, Menlo Park CA 94025
and
Center for the Study of Language and Information
Stanford University, Stanford CA 94305

1 Introduction

The original vision of Logic Programming called for using predicate logic as a programming language [van Emden & Kowalski 76]. Prolog only partially realizes this vision, since it has many features with no corresponding feature in first order predicate logic, and also fails to realize every feature of predicate logic. Perhaps the main benefit of the system suggested in this paper, hereafter called Eqlog, is the way it combines the technology of Prolog (its efficient implementation with unification and backtracing) with functional programming (in an efficient first order rewrite rule implementation) to yield more than just their sum: logical variables can be included in equations, giving the ability to find general solutions to equations over user defined abstract data types (ADTs); this new power is provided in a uniform and rigorous way by using "narrowing" from the theory of rewrite rules to get a complete implementation of equality; it can be seen as a special kind of resolution. In addition, user definable ADTs and generic (i.e., parameterized) modules become available with a rigorous logical foundation; Eqlog also has a subsort facility that greatly increases its expressive power. Since our approach to generic modules and ADTs relies on general results from the theories of specification languages and rewrite rules, it applies to ordinary unsorted Prolog, and should also apply to other logic programming languages such as Concurrent Prolog.

Many other authors have synthesized logic and functional programming. For example, [Kornfeld 83] gives several interesting examples (some of which inspired examples given here), but gives no theoretical justification for his implementation of equality; in fact, it is not complete (i.e., it can sometimes fail to find the right answer when one does exist). Moreover, the ADT and object oriented facilities are less general than might be desired, since neither modularity nor strong typing are provided, and functions are not carefully distinguished from predicates. The Funlog language of [Subrahmanyam & You 84] also has infinite data structures, lazy evaluation, and non-determinism; however, no formal logic is given for these features, either model theoretic or proof theoretic, and Funlog's "semantic unification" algorithm is also incomplete[2]. [Hansson, Haridi & Tarnlund 82] suggest a natural deduction technology to implement a superset of Horn clause logic with equality that includes negation and explicit universal quantifiers; the system also handles infinite data structures by lazy evaluation; however, we are not aware of any formal semantic theory for the language. Finally, [Bellia, Degano & Levi 82] describe FPL, a logic programming notation for what is essentially a functional programming language; a rigorous semantics is given, but it does not support logical variables, or solve systems of equations containing them.

2 The Underlying Logic

First order Horn clause logic without equality underlies ordinary Prolog. But there are many other logics, some of which seem to have distinct advantages. Thus, first order logic with equality supports user definable ADTs; and many-sorted logic gives strong typing. Pure equational logic can also give rise to programming languages. One such language is OBJ [Goguen, Meseguer & Plaisted 82, Goguen & Tardo 79], whose operational semantics interprets equations as rewrite rules; this also supports user definable ADTs.

[1] Supported in part by Office of Naval Research Contract No. N00014-82-C-0333, by National Science Foundation Grant No. MCS8201380, and by a gift from the System Development Foundation to the Center for the Study of Language and Information at Stanford University.

[2] Completeness of the algorithm underlying a logic programming language guarantees that what a user writes will eventually produce the result that the logic says it should.

We now briefly review many-sorted Horn clause logic with equality. Here, one has a set S of **sorts**, plus **signatures** Π and Σ which give the predicate and function symbols, respectively. Each predicate symbol Q has an **arity** which is a string of sorts that serves to indicate the number and sort of arguments that it can take; thus, arity $s_1 s_2 s_1$ indicates that Q takes three arguments, of which the first and third must be of sort s_1, and the second of sort s_2. Similarly, each function symbol has a **rank** consisting of a sort s (its **value** sort) and a string w of sorts (for the sorts of its arguments). Equality enters as a distinguished binary predicate symbol $=_s$ for each sort s, which we will write with infix notation, usually without the subscript. Sentences are Horn clauses in the usual sense, but may involve the distinguished equality predicate; that is, they are of the form

$$P \text{ :- } P_1, \ldots, P_n.$$

where each P and P_i is a positive atomic formula of the form $Q(t_1, \ldots, t_n)$, and each t_i is a term of sort s_i when $s_1 \ldots s_n = w$ is the arity of Q; these terms may include variables, which will of course be "logical variables"; also P and/or any P_i can be equations, since it may use an equality predicate. P is called the **head** of the clause, and $P_1, \ldots, P_n$ constitute its **tail**.

A simple Eqlog[3] program for calculating the population density of countries is

```
density(C) = pop(C) / area(C).
```

In ordinary Prolog, this would be given by the clause

```
density(C,D) :- pop(C,P), area(C,A), D is P / A.
```

using the impure `is` feature, which is a weak analog of Lisp's `eval` function. Also, we can add facts to the database with assertions like

```
pop(china) = 800.
```

(in millions!) instead of the more awkward

```
pop(china,800).
```

Similarly, we can compute the temperature in Fahrenheit from that in Centigrade by the usual formula,

```
f(C) = (9 / 5)* C + 32.
```

where f is a rational (abbreviated **rat**) valued function and C is a **rat** sorted variable (assuming these are available; or, one could use floating point numbers)[4]. However, we can still write the query `f(C) = 77.` and get the right answer `C = 25` (but unless a suitable output simplifier is provided, one is liable to get large unreduced fractions).

[3] We use the convention that variables names begin with a capital letter, while both function and predicate names are all lower case.

[4] Compare this with [Kornfeld 83], which uses functions like **$times** having bizarre definitions that seem to involve putting arbitrary Lisp functions inside clauses.

We now indicate how to get the rationals from the integers by using equality. In fact, one can define equality of rational numbers just as usual in mathematics,

```
X / Y = Z / W :- Y * Z = X * W.
```

where / is a **rat**-valued function symbol denoting division (the denominator must be nonzero), and X, Y, Z, W are variables of sort **Int** (i.e., integer). The above clause (with a little syntactic sugar for declarations, as shown in Section 5) will enable an Eqlog user to define the rationals; by contrast, [Kornfeld 83] uses logical variables in a non-obvious way.

Logical precision requires specifying the intended models. For first order many-sorted logic with equality, these have one set for each sort s, together with a predicate among those sets for each predicate symbol, having arguments of the sorts in its arity; similarly, with a function among those sets corresponding to each function symbol, such that the argument and values match those of the sorts in its rank. It is also assumed that equality predicates are always interpreted as actual equalities in the models. In addition, there may be a number of sorts and associated function and predicate symbols that have a fixed interpretation. For example, it is desirable to build in the integers for reasons of efficiency.

A model M **satisfies** a clause of the form

$$P \text{ :- } P_1, \ldots, P_n.$$

iff for every assignment α of values in the model M to variables in the clause (such that sort restrictions are satisfied), αP holds in M whenever αP_i holds in M for all i. A model M **satisfies** a set C of clauses iff it satisfies every clause in C. However, we are not really interested in *all* models satisfying all the clauses in C; on the contrary, we are only interested in the "standard" model of C, which we now explain. Given signatures Σ and Π of function and predicate symbols (respectively) and a set C of Horn clauses (with equations), the standard model, denoted $T_{\Sigma,\Pi,C}$, has as its elements equivalence classes of ground terms under the equivalence relation

$$t \equiv t' \text{ iff } C \vdash t =_s t',$$

where $\vdash$ is the provability relation for many-sorted first-order logic with equality. Let $[t]$ denote the equivalence class of t under this relation. Then function symbols are interpreted in the usual way, and predicate symbols are interpreted by: $P([t_1],\ldots,[t_n])$ is true in $T_{\Sigma,\Pi,C}$ iff $C \vdash P(t_1,\ldots,t_n)$; and is false otherwise. $T_{\Sigma,\Pi,C}$ is like the Herbrand universe, except that it consists of equivalence classes of terms instead of individual terms.

The basic facts in this situation are given by:

Theorem 1: Let C be a set of Horn clauses with equality, using function and predicate symbols from the signatures Σ and Π respectively. Then:

1. $T_{\Sigma,\Pi,C}$ satisfies C;
2. if M is any other model satisfying C, then there is a unique Σ,Π-homomorphism h: $T_{\Sigma,\Pi,C} \rightarrow M$ (where a Σ,Π-homomorphism is a many-sorted function preserving the function and predicate symbols in the signatures), i.e., $T_{\Sigma,\Pi,C}$ is an **initial** Σ,Π-model satisfying C;
3. any model initial among those satisfying C is isomorphic to $T_{\Sigma,\Pi,C}$;
4. two Σ-terms denote the same element of $T_{\Sigma,\Pi,C}$ iff they can be proved equal using the clauses in C; and
5. for P a predicate symbol and $t_1,...,t_n$ terms in variables $Y_1,...,Y_m$, one has
   $C \vdash (\exists Y_1,...,Y_m)\, P(t_1,...,t_n)$
   iff there is a substitution σ sending the Y_i to ground terms such that $P(|\sigma(t_1)|,...,|\sigma(t_n)|)$ is true in $T_{\Sigma,\Pi,C}$.

□

All this is just another way of stating the so-called "Closed World" assumption for the initial model $T_{\Sigma,\Pi,C}$. This model has "no junk" in the sense that that every element of the model can be denoted by a term using the given function symbols, and "no confusion" in that a predicate holds of some elements iff it can be proved to hold using the axioms; in particular, two elements are identified iff they can be proved equal using the given axioms. In fact, these two conditions together are equivalent to initiality. Note that full first order predicate calculus does not always have initial models in this sense.

3 Solving Equations over Built-in Sorts

Assume that we are given a signature Σ of function symbols and a reachable Σ-model A.[5] Now let E be a set of Σ-equations over a set X of variables. Then a **ground solution** of E in A is an assignment α from the variables in X to values in A such that $\alpha(E)$ is satisfied in A. Now letting $T_\Sigma(Y)$ denote the Σ-terms with variables from Y, we define a **solution** of E in A to be an assignment σ from X to terms in $T_\Sigma(Y)$ such that $\alpha(\sigma(E))$ is satisfied in A for every assignment α from Y to A. A **complete solution** of E in A is a set L of solutions such that every solution of E in A is a substitution instance of one in L; i.e., such that for any solution τ (from variables X to $T_\Sigma(Y)$) there is a solution σ in L and a substitution ρ from the variables in Y to $T_\Sigma(Y)$ such that $\tau = \rho(\sigma)$. (Note that these definitions do not require most general substitutions.)

For example, let **N** be the natural numbers with only the function +, so that Σ contains elements of **N** as constants and +. Let us consider just linear equations, regarding 3X as an abbreviation for (X + X + X). Thus, the equations

$$3X + Y + 2Z = 1$$
$$X - 2Y = 3$$

has a ground solution $\sigma(X)=7$, $\sigma(Y)=2$, $\sigma(Z)=-11$, and has a complete solution given by $\sigma(X)=3+4V$, $\sigma(Y)=2V$, $\sigma(Z)=-4-7V$, where V is a parameter variable. It is a general theorem that any set of linear equations over the integers has either no solution, or else a complete solution consisting of just one substitution.

Complete solutions do not necessarily exist; also, just because a complete solution exists does not mean that it is recursively enumerable, i.e., that there is an algorithm that will produce all the substitutions in it. Moreover, even if a recursively enumerable complete solution exists, the algorithm can still fail to terminate when faced with a case for which no solution exists. Let us say that we have a **totally complete solution** in case there is an algorithm that will explicitly fail if there is no solution, and otherwise will enumerate a complete solution. Similarly, let us say we have a **r.e. complete solution** in case there is an algorithm that will enumerate a complete solution when there is one, and say we have a **finite solution** if we have a totally complete solution that is always finite. More algorithmically, we will assume that SOLN(E) produces substitutions in the solution of E, if any exist, one at a time on request until there are no more.

A further desirable property of a solution L of E in A is that it should be **most general**, in the sense that for any solution substitution σ, there is a unique member τ of L and a unique substitution ρ such that $\rho\tau = \sigma$. It can be shown that any two most general solutions are essentially the same. Unfortunately, there are cases where totally complete solutions exist, but no most general solution exists. The classical case is where the model is the set of terms over some signature Σ, and the functions are just those in Σ. Then unification gives a finite solution (totally complete, with just one most general unifier).

[5] This means that every element of A is denoted by some Σ-ground term.

4 Computing in Horn Clause Logic with Equality

This section considers sublogics of Horn clause logic with equality within which equations over user definable ADTs can often be solved. We begin with a basic logic and then extend it; most logic programming applications seem to be included. The basic sublogic assumes all clauses are of two types, either a pure equation, or else a clause whose head is not an equation. Let $\mathcal{E}$ denote the set of equations and P the set of Horn clauses whose head clause is not an equation; thus, $C = \mathcal{E} \cup P$. To unify two positive atomic formulae, say $Q_1(t_1,...,t_n)$ and $Q_2(u_1,...,u_m)$, we must of course have that Q_1 is Q_2, the arity w_1 of Q_1 is the arity w_2 of Q_2 so that n=m and the sort of t_i equals that of u_i, and we must also solve the system

$t_1 = u_1, ..., t_n = u_n$

of simultaneous equations modulo the equations given in $\mathcal{E}$; this is called **$\mathcal{E}$-unification**. Because of our assumptions about the structure of clauses, those in P can have no influence on $\mathcal{E}$-unifiers.

The computation algorithm of ordinary Prolog has been described clearly but informally by [Warren 80]: "To *execute* a goal, the system searches for the first clause whose head *matches* or *unifies* with the goal. The *unification* process finds the most general common instance of the two terms, which is unique if it exists. If a match is found, the matching clause instance is then *activated* by executing in turn, from left to right, each of the goals of the body (if any). If at any time the system fails to find a match for a goal, it *backtracks*, i.e., it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. Next it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal."

When unification is attempted, Eqlog must also call SOLN, in a way that permits backtracking and is fair, in that every substitution in SOLN(E) gets tried (this is a semidecision procedure and may not halt; but when SOLN is r.e. complete, then our algorithm is complete). Say that a predicate P (which may be an equality $=_s$) **directly depends** on another Q if there is a clause with P as the predicate of its head and Q as a predicate in its tail; let **depends** be the transitive closure of direct dependence. Then we conjecture that our evaluation algorithm works provided no equality predicate depends on itself. For example, it is reasonable to define $=_{rat}$ in terms of $=_{int}$ since there is no dependence of the clauses defining Int on those defining **rat**.

5 User Defined Abstract Data Types

There is much work on providing user defined ADTs in programming languages (see Clu and Ada) and on the foundations in equational logic (e.g., [Meseguer & Goguen 84, Goguen, Thatcher & Wagner 78]). The essential idea is that users introduce modules that define new sorts and associated functions. A purely syntactic notion of module has been given for Mprolog by [Domolki & Szeredi 83].

Let us now give a complete definition for the data type **rat** in proper Eqlog syntax. Eqlog keywords are underlined, and module names are in capitals (built-in types come in modules; the module INT has sort Int with subsort nzInt of nonzero integers). "Attributes" can be given for operators; for example, assoc, comm, and idp indicate that a binary operator is associative, commutative, and idempotent, respectively; and id: e indicates that it has e as its identity. The associative and commutative properties of functions can be built into unification algorithms. Eqlog "mix-fix" notation permits any desired ordering of keywords and arguments for operators; this is declared by giving a syntactic "form" consisting of a string of keywords and underbar characters (_), followed by a ":", followed by the arity as a string of sorts, followed by "->", followed by the value sort of the function; if there are no underbars, then the usual parentheses with comma notation must be used. Similar conventions are used for predicates. An expression is considered "well-formed" in this scheme iff it has exactly one parse; the parser can interactively help the user to satisfy this condition[6]

```
module BASICRAT using INT is
 sorts rat
 subsorts Int < rat
 fns
   _/_ : Int,nzInt -> rat
   _*_ : rat,rat -> rat (assoc comm id: 0)
   _+_ : rat,rat -> rat (assoc comm id: 1)
 vars X,Y,Z,W,N : Int
 axioms
   N = N / 1.
```

[6] The parser is greatly helped if spaces always separate the keywords declared in the form of a function, and this paper follows that convention throughout; but since parentheses are also delimiters, they do need not to be separated by spaces. These syntactic conventions follow those of OBJ [Goguen, Meseguer & Plaisted 82].

```
  X / Y = Z / W :- X * W = Y * Z.
  (X / Y)*(Z / W) = (X * Z)/(Y * W).
  (X / Y)+(Z / W) =
      ((X * W) + (Z * Y))/(Y * W).
endmod BASICRAT
```

Here the keyword using indicates that the sorts, subsorts, predicates, functions, and axioms of the listed modules should be imported to the module being defined. We will refer to the relationship between modules being defined and being used as the **using hierarchy**. We now enrich BASICRAT to define division and the subsort of nonzero rationals.

```
module RAT using RATO is
 sorts nzrat
 subsorts nzrat < rat
 fns
     _/_ : rat,nzrat -> rat
 vars X,Y,Z,W : Int
 axioms
     nzrat(X / Y) :- nzint(X).
     (X / Y)/(Z / W) = (X * W)/(Y * Z).
endmod RAT
```

We have already noted that the sorts and subsorts currently defined form an acyclic graph (thus supporting so-called "multiple inheritance"). This motif is repeated at the module level, with another acyclic graph under the using hierarchy. In fact, the subsort hierarchy and the using hierarchy interact, since subsorts are declared inside of modules: At a given node M of the using hierarchy, the set of curently defined sorts is the union of those declared in M with all those declared in nodes below M in the using hierarchy (i.e., all those related to M by the transitive extension of the using relation); similarly, the subsort relation active at M is the union of the subsort declarations in M with those from modules below M. Thus, the subsort graph of a lower level module is a subgraph of that of a higher level module. (All this has already been implemented in OBJ and has been found very natural and helpful.)

6 Generic Modules

Reusability is a major goal of modern software engineering. In order to achieve this goal, it is necessary that software be broken into components that are as reusable as possible; parameterization is a technique that can greatly enhance the reusability of components [Goguen 83]. For example, bag-of and set-of, which have caused considerable controversy in the *Prolog Digest*, can easily be defined as generic abstract data types, and then automatically implemented using rewrite rules. Generic modules also greatly ameliorate the otherwise odious need for defining abstractions whenever they are used.

Before giving details, we consider how to specify a parameterized module's interface, especially the **requirements** that an actual parameter should satisfy for the instantiation to make sense, expressed in the form of a theory, that is, a set of axioms, that the actual must satisfy. Such a theory may include sort, subsort, predicate and function declarations, saying what the actual parameter must provide to the parameterized module, as well as axioms saying what properties must be satisfied. For example, a generic sorting module might have the theory of quasi-ordered sets as its requirement theory; this means that an actual must provide a designated sort and a binary relation on it that is transitive and reflexive. In Eqlog, this theory is given as follows:

```
theory QUOSET is
   sorts elt
   preds _=<_ : elt,elt
   vars A,B,C : elt
   axioms
      A =< A.
      A =< C :- A =< B, B =< C.
endth QUOSET
```

Theories are not intended to be used for computation, but only for declaring the properties of interfaces. The idea is that before an instantiation of a generic can be "certified," it must be shown that the actual parameter does in fact have the properties required by the theory. Because computations do not use the axioms given in theories, there is no reason to restrict the form of the axioms in theories, and in fact, we allow arbitrary first order axioms. Difficulty only arises when one has to prove that the axioms hold of some particular module; then, one needs a first order theorem prover. Here is an even simpler theory, the one that is actually used for the generic SET example:

```
theory TRIV is
   sorts elt
endth TRIV
```

This theory requires nothing except that a particular sort be designated. We now give a generic BASICSET module, providing only symmetric difference, $\uplus$, and intersection; later we will define the rest of the set functions from these. After the name of the module comes a left square bracket, indicating that what follows is the formal parameter symbol, P in this case, and after the :: comes the theory that it is required to satisfy; the formal parameter part is then closed by a right square bracket.

```
module BASICSET[ELT :: TRIV] is
```

```
sorts set
fns
 ∅,Ω : set
 {_} : elt -> set
 _⊎_ : set,set -> set (assoc comm id: ∅)
 _∩_ : set,set -> set (assoc comm idp
     id: Ω)
vars S,S',S'' : set, elt,elt' : elt
axioms
 S ⊎ S = ∅.
 { elt } ∩ { elt' } = ∅ :- elt ≠ elt'.
 S ∩ ∅ = ∅.
 S ∩(S' ⊎ S'') = (S ∪ S')⊎(S ∪ S'').
endmod BASICSET
```

This way of defining finite sets follows [Hsiang 81]'s approach to the propositional calculus; Ω is the "universal" set, i.e., the set of all things of sort **elt**. The attribute `id:` should be taken as an abbreviation for the identity equation. In many cases, this definition will execute faster than more conventional axiomatizations. It should be noted that the BASICSET module provides not only all finite subsets of the set given as actual parameter, but also all cofinite sets (i.e., sets whose complement is finite). The inequality in the axiom

```
{ elt } ∩ { elt' } = ∅ :- elt ≠ elt'.
```

violates the purity of the language only in appearance, since Section 7 shows how to reduce the semantics of inequality to that of equality.

To instantiate a generic module, one must provide an actual parameter A; but more than this is needed. Since both modules and theories can involve more than one sort, we need to say just which sorts in the actual correspond to those declared in the requirement theory T of the generic; similarly, we need to say which functions and predicates in an actual A correspond to those required by the theory. Following [Goguen 83] and ideas from Clear, this correspondence is given by a **view**, which consists of:

1. a function from the sorts of the theory T to those of A;
2. a function from the functions of T to those A; and
3. a function from the predicates of T to those of A,

such that

- the subsort relation is preserved;
- the sorts of functions and predicates are preserved; and
- the translations of the axioms in T to axioms about A are in fact true of the initial model of A.[7]

In the language of [Goguen & Burstall 84], a view is a "theory morphism."

In many cases, it is obvious how to construct a view of A as T; this is formalized by the notion of a **default view** in [Goguen 83]. In other cases, there is only one appropriate view in the current environment, and of course that is the one to apply. In such cases, it is not necessary to indicate what view is intended, one can just write the name of the actual. For example, in order to construct SET-OF-INT, we just say

```
make SET-OF-INT is SET[INT] endmake
```

since there is a default view of INT as a TRIV. In other cases, it may be necessary to include a view in the `make` statement. For example,

```
make SORTING-OF-INT-DIV is SORTING[INT-
AS-DIV-QUOSET] endmake
```

instantiates a generic SORTING module with the quoset of integers ordered by the divisibility relation. When it is not necessary to give the instantiated module a name, we can just write, e.g., `SET[INT]`.

We now enrich the generic BASICSET module given earlier (recall that it provided symmetric difference and intersection) to provide union, difference and cardinality functions, plus some of the usual predicates.

```
module SET[X :: TRIV] using NAT,
   BASICSET[X] is
 fns
   _∪_ : set,set -> set
   _-_ : set,set -> set
   #_ : set -> nat
 preds
   _∈_ : elt,set
   empty : set
   _∉_ : elt,set
 vars X : elt, S,S',S'' : set
 axioms
   S ∪ S' = (S ∩ S')⊎ S ⊎ S'.
   S - S' = S ∩(S ⊎ S').
   empty(S) :- S = ∅.
   N ∈ S :- { N } ∪ S = S.
   N ∉ S :- { N } ∩ S = ∅.
   # ∅ = 0.
   #({ X } ∪ S) = succ(# S) :- X ∉ S.
   #({ X } ∪ S) = # S :- X ∈ S.
endmod SET
```

Although # does not yield the answer ∞ for infinite sets, it does work reasonably. For example in the case of `SET[INT]`, # Ω is just # Ω

[7] In practical large scale programming, one may wish to settle for less than a formal proof of this; for example, an informal proof might be acceptable.

again, a reduced term rather than a non-terminating computation. Also, #({ 14 } ∪ Ω) evaluates to # Ω again.

We can also enrich a module without giving the enrichment an explicit name; this can be useful if some constants are being defined for a single query or example. Another feature illustrated by the following module is that when the requirement theory is TRIV, a view can be determined just by giving a sort name (provided that the sort only occurs in one module in the current environment). If the sort name does not occur in any module in the current environment, then it serves to declare a new sort and apply the generic to it; we shall call this a declaration "on the fly."

Views also provide an elegant form of declaration at the module level. In ordinary sequential programming, "assertions" can be inserted after a statement to indicate that the program's state is supposed to satisfy some property after the execution of that statement. In logic programming with modules, a view from a theory to a module serves to indicate that the module (i.e., its sorts, functions and predicates) satisfies certain axioms. It should be noted that one can also compose generics. For example, one can form BAG[SET[INT]].

Of course, there is nothing special about the details of the features and syntax described here for Eqlog modules and generics; what is special is the underlying semantic ideas. Unfortunately, there is not room in this paper for a full exposition of this semantics, which is based on ideas from the Clear specification language [Burstall & Goguen 80]. The ideas are not really difficult, but they use some comparatively advanced mathematics. Some discussion of the issues involved is given in Section 7. The application of these ideas to the equational logic programming language OBJ is described in [Goguen 83].

7 Logical Foundations

This section discusses in more detail four issues regarding the foundations of Eqlog: subsorts, institutions, narrowing, and inequality.

7.1 Subsorts and Institutions

Many of our examples use subsorts and subsort predicates. We now explain why this is not an impure feature, but rather an expressive shorthand for a specification in standard Horn clauses logic with equality. We also describe conditions that insure valid use of the equality predicate; these conditions could be enforced syntactically. Although more permissive uses of subsort predicates are possible and certainly worth exploring, the one presented here is already very general.

Whenever a subsort $s < s'$ is declared, a corresponding unary predicate $s(\_)$ of sort s' also becomes available; intuitively, this predicate is true of a term iff that term lies in the subsort. Users can give axioms involving the subsort predicate; but these should only assert that certain functions restrict (and constants belong) to the subsort. For example, the subsort pos < Int of positive integers can be characterized as containing 1 = succ(0) and being closed under the successor function, by the two clauses

```
pos(1).
pos(succ(X)) :- pos(X).
```

Our reconstruction of subsorts within Horn clause logic with equality involves giving ordinary signatures Σ and Π, and a set C of Horn clauses, such that the initial model $T_{\Sigma,\Pi,C}$ is isomorphic to the model intended for the subsort declarations and their corresponding predicates. The first step is to introduce a new ordinary sort for each subsort. We then force that in all models, the new sort s is identified with a subset of the sort s' whenever $s < s'$ by introducing a new function symbol j : s -> s' that is made to play the role of an inclusion by satisfying the axiom

```
j(X) = j(Y) :- X = Y.
```

Similarly, we can express the fact that certain functions or constants restrict to a subsort by introducing new function symbols for these functions and constants such that their value sort is the subsort; equations are then given to insure their relationship to the functions and constants in the supersort.

The module, theory, view and instantiation features of Eqlog support generic (i.e., parameterized) programming, a form of programming-in-the-large that seems to permit an unusually high degree of reusability. All these features can be defined for any logical system satisfying some very simple and reasonable axioms that make it an **institution** [Goguen & Burstall 84]. In particular, it has been shown that the logic of Horn clauses with equality is an institution, so the general machinery can be applied directly to this case, giving a semantics for the parameterization features in Eqlog. There is not room here for the details of this approach, which relies on category theoretic concepts like colimit. It is worth remarking that the subsystem of Horn clause logic with equality consisting of pure equations plus Horn clauses whose heads are not equations, is also an institution.

7.2 Unification in an Equational Theory

An equational theory is given by a pair (Σ,T) where Σ is an S-sorted signature of function symbols and T is a set of Σ-equations. The rules of many-sorted equational deduction [Goguen & Meseguer 81] define an equivalence relation $=_T$ between Σ-terms with variables, namely that of being provably equal using the equations in T. If X denotes an S-sorted set containing an infinite supply of variables of each sort, and if $T_\Sigma(X)$ stands for the Σ-algebra of terms with variables in X, then a **substitution** is an S-sorted function $\alpha: X \to T_\Sigma(X)$; such a function extends to a unique Σ-homomorphism from $T_\Sigma(X)$ to itself that we also denote by α. A substitution α is said to have **domain** $Y=\{Y_s\}$ when $Y_s=\{x\in X \mid \alpha(x)\neq x\}$; we then write $Y=\mathrm{dom}(\alpha)$. The set of variables **introduced** by α is the S-sorted $\mathrm{int}(\alpha)_s=\cup\{\mathrm{vars}(\alpha(x)) \mid x\in\mathrm{dom}(\alpha)_s\}$, where vars(t) denotes the set of variables occurring in a term t. Given an S-sorted set of variables $Y\subseteq X$ and substitutions α and β, we write $\alpha=_T\beta$ [Y] iff $\alpha(x)=_T\beta(x)$ for each x in Y. Similarly, we write $\alpha\leq_T\beta$ [Y] iff there is a substitution γ such that $\beta=_T\gamma\circ\alpha$ [Y]. A **T-unifier** of two terms t and t′ is a substitution α such that $\alpha(t)=_T\alpha(t')$. Given terms t and t′ with $Y=\mathrm{vars}(t)\cup\mathrm{vars}(t')$, a set L of T-unifiers of t and t′ is called a **complete set of T-unifiers** of t and t′ iff for each T-unifier γ of t and t′ there is an α in L with $\alpha\leq_T\gamma$ [Y]. (This was called a most general complete solution in Section 3.) Without loss of generality we may assume, for technical reasons, that $\mathrm{dom}(\alpha)\subseteq Y$ and $\mathrm{int}(\alpha)\cap Y=\emptyset$ for each α in L.

Given an equational theory T, a **complete T-unification algorithm** SOLN is an algorithm such that if started with any two terms t and t′, SOLN generates a complete set of T-unifiers for t and t′; SOLN is **finite** if, in addition, it always terminates with a finite set. Particular unification algorithms for theories T of frequent use, such as associativity, commutativity, etc., have been given in the literature. For the general case when T consists of a confluent and terminating set $\mathcal{R}$ of rewrite rules, a unification algorithm using narrowing has been given by [Fay 79] and improved in order to give a termination criterion by [Hullot 80]. Then, for any two terms one has $t=_Tt'$ iff $\mathrm{can}(t)=\mathrm{can}(t')$, where can(t) is the canonical form of the term t, obtained after exhaustive rewriting by applications of the rules $\mathcal{R}$.

The one step narrowing relation is defined as follows: Let t be a term; by renaming of variables (or some other convention) we can always assume that the variables occurring in t do not occur in any of the rules. Let t_0 be a nonvariable subterm of t that unifies (in the ordinary sense) with the left hand side t_1 of a rule $t_1=t_2$ in $\mathcal{R}$, with α the most general unifier. Let t′ be the term obtained by replacing in $\alpha(t)$ the subterm $\alpha(t_0)=\alpha(t_1)$ by $\alpha(t_2)$. Then we say that t′ is a **one step narrowing** of t, and we write $t\Rightarrow t'$. The **narrowing** relation is the reflexive and transitive closure of one step narrowing, and contains the rewriting relation as a subset. The following algorithm then provides a complete set of T-unifiers.

Theorem 2: [Fay 79, Hullot 80]. Let $T=\mathcal{R}$ be a confluent and terminating set of rewrite rules. Given a pair t,t′ of terms, introduce a new function symbol[8] τ and consider all the narrowing chains that begin with $\tau(t,t')$. If such a chain ends with a term of the form $\tau(t_n,t'_n)$ such that t_n and t'_n are unifiable by a substitution α, then compose α with the substitutions obtained at the previous narrowing steps in the chain, and add this composition to the set of unifiers already generated. The set so obtained is a complete set of T-unifiers for t and t′. □

This algorithm has been extended to handle the more general situation when the equations in T can be partitioned into a set of rewrite rules $\mathcal{R}$ and a set of equations $\mathcal{E}$ in such a way that $\mathcal{R}$ is terminating and confluent *modulo $\mathcal{E}$*. Many common examples fall into this category. A general answer is given by [Jouannaud, Kirchner & Kirchner 83], who generalize Theorem 2 by showing that if there is a finite $\mathcal{E}$-unification algorithm, then narrowing modulo $\mathcal{E}$ still provides a complete $T=\mathcal{R}\cup\mathcal{E}$-unification algorithm. The idea, in this case, is to have part of the T-unification work done by a built-in $\mathcal{E}$-unification algorithm, and the rest by $\mathcal{E}$-narrowing. Both [Hullot 80] and [Jouannaud, Kirchner & Kirchner 83] give sufficient conditions for termination of their algorithms.

Now a simple example showing how a query involving an equation is evaluated by narrowing; for illustrative purposes, this example does not use the built-in natural number type, but rather

[8] The reader may find it helpful to construe this symbol as a formal equality symbol.

provides its own, of sort **nat1**, with successor function succ; also, notice there is no nil list here.

```
module LIST[ELT :: TRIV] is
 sorts elt, list, nat1
 subsorts elt < list
 fns
   0 : nat1
   succ : nat1 -> nat1
   _*_ : list,elt -> list
   length : list -> nat1
 vars Elm : elt, Lst : list
 axioms
   length(Elm) = succ(0).
   length(Lst * Elm) = succ(length(Lst)).
endmod LIST
```

The sort **elt** is a parameter, and is empty in the Herbrand universe; however this causes no problem if a suitable modification of the rules of deduction is used (see [Goguen & Meseguer 81] for the equational case). The query

```
length(Lst') = succ(succ(succ(0))).
```

evaluates to

```
length((Elm'' * Elm') * Elm) =
  succ(succ(succ(0)))
```

by accumulating the substitutions associated with the narrowings from the root `length(Lst')` to the expression `succ(succ(succ(0)))`.

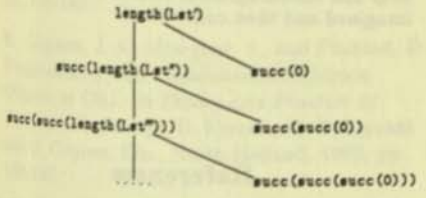


Figure 1: Narrowing on the Length Function

7.3 Equality and Inequality

The use of negation for arbitrary predicates gives rise to difficulties. However, perhaps surprisingly, it is not so difficult to treat the negation of equality. For example, the BASICSET module of Section 6 contains the axiom

$$\{ \text{elt} \} \cap \{ \text{elt}' \} = \emptyset \text{ :- elt} \neq \text{elt}',$$

which appears to lie outside the realm of Horn clause logic with equality. However, this is only an appearance, because the semantics of inequality can be reduced to that of equality. The equational part of any Eqlog module should be a computable abstract data type. This is implicit in our requirement that the equations form a confluent and terminating set of rewrite rules (perhaps modulo some decidable equations such as associativity, commutativity, etc.) since is has been shown that any computable data type can be presented that way. Equality and inequality of ground terms is then built in, since one can just compute the canonical forms of the terms in question and see whether or not they are equal. Moreover, as shown in [Meseguer & Goguen 84], a data type is computable if and only if its equality is finitely axiomatizable by equations. This means that we can always axiomatize equality for each sort **s** as a function `_≡_ : s,s -> bool`, by means of a finite set of equations. **bool** is a new sort having two constants, `true` and `false`, such that for any two ground terms t,t' we have $t=t'$ (in the data type) iff $(t\equiv t')=$`true` (in the equational equality enrichment) and similarly, $t\neq t'$ (in the data type) iff $(t\equiv t')=$`false` (in the equational equality enrichment). In this way, inequality is reduced to equality.

Given an inequality $t \neq t'$, the Eqlog system will then:

1. compute it by rewriting if both t and t' are ground terms; and
2. otherwise, requiring the existence of an equationally defined equality, $\equiv$, for the sort in question, translate the inequality into the equation $(t\equiv t') =$ `false`, and then solve this equation using narrowing.

8 The Missionaries and Cannibals Problem

To illustrate the power of Eqlog, we show how to use some standard generics, plus subsorts, functions and predicates, for a general Missionaries and Cannibals problem (hereafter, MAC); once the parameters are instantiated, Eqlog solves MAC by $\mathcal{E}$-narrowing, for $\mathcal{E}$ a set of equations including associativity and commutativity equations for the set operations. We begin with a theory MACTH of the preconditions for MAC: there are two disjoint sets of persons, m0 of missionaries and c0 of cannibals. Later we instantiate MACTH to the usual case of three missionaries and three cannibals. MACTH uses a generic SET module to get set difference, union, and cardinality. By convention, a module with a "principal" sort has the same name as that sort (unless explicitly indicated otherwise); e.g., the sort of PSET is **pset**.

```
theory MACTH[PERSON :: TRIV] using SET,
      PSET = SET[PERSON] is
  fns
     m0 : pset
     c0 : pset
  axioms
     m0 ∩ c0 = ∅.
endth MACTH
```

The MAC module also uses a generic LIST module that provides the empty list nil, the length function #, and concatenation *. The new sort **trip** is introduced "on the fly" (see Section 6) in the submodule TRIPLIST. We now briefly discuss the intuition behind this specification. A solution is a list of trips having certain "good" properties, where a trip is a boat containing a set of persons; odd numbered trips go from the left bank to the right, and even trips go from the right to the left. Missionaries and cannibals are persons. The predicate boatok indicates that a boat has an ok number of persons; the predicate good is true if a list of trips never allows there to be more cannibals than missionaries on a bank; the predicate solve indicates that a trip list is a solution to the problem. The functions lb and rb give the sets of persons on the left and right banks, respectively, and the functions mset and cset extract the subsets of missionaries and cannibals (respectively) from a set of persons.

```
module MAC[T :: MACTH] using NAT,
      TRIPLIST = LIST[trip] is
 preds
   boatok : trip
   solve,good : triplist
 fns
   boat : pset -> trip
   lb,rb : triplist -> pset
   mset,cset : pset -> pset
 vars PS : pset, L : triplist,
      P : person, T : trip
 axioms
   boatok(boat(PS)) :- # PS = 1.
   boatok(boat(PS)) :- # PS = 2.
   lb(nil) = m0 ∪c0.
   mset(PS) = PS ∩ m0.
   cset(PS) = PS ∩ c0.
   rb(nil) = ∅.
   lb(L * boat(PS)) = lb(L) - PS :-
      even # L.
   rb(L * boat(PS)) = rb(L) ∪ PS :-
      even # L.
   rb(L * boat(PS)) = rb(L) - PS :-
      odd # L.
   lb(L * boat(PS)) = lb(L) ∪ PS :-
      odd # L.
   good(L * T) :- # cset(lb(L * T)) =<
      # mset(lb(L * T)),
      # cset(rb(L * T)) =<
      # mset(rb(L * T)), good(L),
      boatok(T).
   good(nil).
   solve(L) :- good(L), lb(L) = ∅.
endmod MAC
```

Now the constants to instantiate MAC to the usual case.

```
module EX1 using SET[ID] is
  axioms
     m0 = { taylor, helen, willian }.
     c0 = { umugu, nzvave, anoc }.
endmodule EX1
```

The notation {a, b, c} is shorthand for { a } ∪ { b } ∪ { c }. We can now instantiate MAC and ask Eqlog to solve the resulting problem with

```
make MAC[EX1] endmake
solve(L.)
```

using the default view of EX1 as MACTH, and not bothering to give the resulting module a name.

Acknowledgements

We extend our most sincere thanks to Jean-Pierre Jouannaud and Fernando Pereira for their extensive comments on this paper, and for their help and encouragement while it was being imagined and then constructed.

References

1. Bellia, M., Degano, P. and Levi, G. The Call by Name Semantics of a Clause Language with Functions. In *Logic Programming*, Clark, K.L. and Tarnlund, S.-A., Eds., Academic Press, 1982, pp. 281-295.

2. Burstall, R. M., and Goguen, J. A. The Semantics of Clear, a Specification Language. In *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, Springer-Verlag, 1980, pp. 292-332.

3. Domolki, B. and Szeredi, P. Prolog in Practice. In *Information Processing 83*, Mason, R. E. A., Ed.,Elsevier, 1983, pp. 627-636.

4. Fay, M. "First-order Unification in an Equational Theory." *Proceedings, Fourth Workshop on Automated Deduction 4* (February 1979), 161-167.

5. Goguen, J. A. Parameterized Programming. In *Proceedings, Workshop on Reusability in Programming*, Biggerstaff, T. and Cheatham, T., Eds., ITT, 1983, pp. 138-150.

6. Goguen, J. A. and Burstall, R. M. Introducing Institutions. In *Proceedings, Logics of Programming Workshop*, E. Clarke and D. Kozen, Ed.,Springer-Verlag, 1984, pp. 221-256.

7. Goguen, J. A. and Meseguer, J. "Completeness of Many-sorted Equational Logic." *SIGPLAN Notices 16*, 7 (July 1981), 24-32. Also appeared in *SIGPLAN Notices*, January 1982, vol. 17, no. 1, pages 9-17; extended version as SRI Technical Report, 1982, and to be published in *Houston Journal of Mathematics*.

8. Goguen, J. A. and Tardo, J. An Introduction to OBJ: A Language for Writing and Testing Software Specifications. In *Specification of Reliable Software*, IEEE, 1979, pp. 170-189.

9. Goguen, J. A., Meseguer, J., and Plaisted, D. Programming with Parameterized Abtract Objects in OBJ. In *Theory and Practice of Software Technology*, D. Ferrari, M. Bolognani and J. Goguen, Eds., North-Holland, 1982, pp. 163-193.

10. Goguen, J. A., Thatcher, J. W. and Wagner, E. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*, R. Yeh, Ed.,Prentice-Hall, 1978, pp. 80-149.

11. Hansson, A., Haridi, S. and Tarnlund, S.-A. Properties of a Logic Programming Language. In *Logic Programming*, Clark, K.L. and Tarnlund, S.-A., Eds., Academic Press, 1982, pp. 267-280.

12. Hsiang, J. Refutational Theorem Proving using Term Rewriting Systems. Ph.D. Thesis, Univeristy of Illinois at Champaign-Urbana.

13. Hullot, J.-M. Canonical Forms and Unification. In *Proceedings, 5th Conference on Automated Deduction*, W. Bibel and R. Kowalski, Eds., Springer-Verlag, Lecture Notes in Computer Science, Volume 87, 1980, pp. 318-334.

14. Jouannaud, J.-P., Kirchner, C., Kirchner, H. Incremental Construction of Unification Algorithms in Equational Theories. Automata, Languages and Programming, Barcelona, 1983., 1983, pp. 361-373.

15. Kornfeld, W. A. "Equality for Prolog." *Proceedings, Seventh International Joint Conference on Artificial Intelligence 7* (1983), 514-519.

16. Meseguer, J. and Goguen, J. A. Initiality, Induction and Computability. In *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, Eds., Cambridge University Press, 1984.

17. Subrahmanyam, P. A. and You, J.-H. Pattern Driven Lazy Reduction: a Unifying Evaluation Mechanism for Functional and Logic Programs. In *Proceedings, Eleventh ACM Symposium on Principles of Programming Langagues*, ACM, 1984, pp. 228-234.

18. van Emden, M. H. and Kowalski, R. A. "The Semantics of Predicate Logic as a Programming Language." *Journal of the Association for Computing Machinery 23*, 4 (1976), 733-742.

19. Warren, D. "Logic Programming and Compiler Writing." *Software - Practice and Experience 10* (1980), 97-125.

UNFOLD/FOLD TRANSFORMATION OF LOGIC PROGRAMS

Hisao TAMAKI
Ibaraki University
Dept. of Information Science
Hitachi, 316, Japan

Taisuke SATO
Electrotechnical Laboratory
Machine Inference Section
Umezono, Sakura-mura, 305
Japan

ABSTRACT

The unfold/fold transformation method is formulated for logic programs in such a way that the transformation always preserves the equivalence of programs as defined by the least model semantics. A detailed proof for the basic system is presented first. Then some augmenting rules are introduced and the conditions of their safe application within the unfold/fold system are clarified. There are useful special cases of those rules whose application is always safe.

1 INTRODUCTION

The unfold/fold program transformation method was developed by Burstall and Darlington (Burstall & Darlington 1977) in the context of their recursive equation language. The idea was generalized and applied to logic program synthesis(Clark & Sickel 1977)(Hogger 1981), where the authors naturally formulated the unfold and fold transformations as just special cases of logical deduction. Thus each clause in the synthesized program is a theorem deduced from the specification axioms. This ensures the partial correctness of the synthesized program because every result of computation (atomic theorem deduced from the program) is derivable directly from the specification as well. Total correctness, however, is not guaranteed in general and should be proved separately(Clark 1979).

They applied this deductive approach to logic program transformation taking the initial program, viewed as if-and-only-if definitions, to be the specification. But what is ensured in general is again just partial correctness: the relations computed by the transformed program are narrower or equal to those computed by the original one. In other words the least Herbrand model(Van Emden & Kowalski 1976) of the transformed program is included in that of the initial one. If we want exact equivalence, the inverse inclusion should be proved for individual cases.

As an alternative to the deductive approach, we have formulated an unfold/fold transformation system for logic programs(Tamaki & Sato 1983) in such a way that the transformation always preserves the equivalence of programs as defined by the least model semantics. Though we have to sacrifice the generality of the deductive approach, the guaranteed equivalence should worth the cost.

This paper augments the basic unfold/fold system with

some other transformation rules. Though the rules themselves are obviously equivalence preserving, their interaction with unfold/fold transformation needs careful study. The condition for the application of the rules to be safe will be clarified.

Section 2 describes the basic unfold/fold system and proves that it preserves the equivalence of programs. The proof is simpler than the one given in (Tamaki & Sato 1983) and more suitable for our purpose. Section 3 and 4 introduce and study augmenting rules.

The readers are assumed to be familiar with the standard notions and notations of logic programs (Kowalski 1974). Note that our target language is a pure one rather than a practical implementation such as existing Prologs. Thus a *program* is a set (not an ordered list) of definite clauses. A *definite clause* is a pair of a goal(atomic formula), called a *head*, and a multi-set (again not an ordered list) of goals called a *body*.

2 BASIC UNFOLD/FOLD SYSTEM

2.1 Description of the system

The transformation process proceeds as follows.

Transformation process

```
begin P0 := the initial program;
      D0 := {}; /* the set of
                   definitions of
                   new predicates */
      mark every clause in P0
      'foldable';
      for i := 1 to arbitrary N
          apply any of the trans-
          formation rules to ob-
          tain Pi and Di from
          Di-1
end
```

In this section we are only concerned with the three basic rules, namely, *definition*, *unfolding* and *folding*, each of which are described in the sequel.

Example (initial program)

P_0 : C1. subseq([],X)

C2. subseq([A|X],[A|Y]) ← subseq(X,Y)

C3. subseq(X,[A|Y]) ← subseq(X,Y)

We use this example to illustrate the process and rules of transformation. The upper case letters are variables, [] denotes an empty list and [A|X] a list with head A and tail X. Thus the predicate subseq(X,Y) is intended to mean that X is a subsequence of Y.

Rule 1. definition

Let C be a clause of the form $p(x_1,\ldots,x_n) \leftarrow A_1,\ldots,A_m$, where

1. p is an arbitrary predicate not appearing in P_{i-1} or D_{i-1},
2. $x_1,\ldots,x_n$ are distinct variables, and
3. $A_1,\ldots,A_m$ are goals whose predicates all appear in P_0.

Then let P_i be $P_{i-1} \cup \{C\}$ and D_i be $D_{i-1} \cup \{C\}$.

Do not mark C 'foldable'.

The predicates introduced by the definition rule are called *new* predicates while those in P_0 are called *old*. Those variables occurring in $A_1,\ldots,A_m$ other than $x_1,\ldots,x_n$ are called *internal variables* of C.

Example (continued)

We define C4, motivated by some need for a common subsequence relation.

C4. csub(X,Y,Z) ← subseq(X,Y), subseq(X,Z)

Then $P_1 = \{\underline{C1},\underline{C2},\underline{C3},C4\}$, $D_1 = \{C4\}$. Underline indicates 'foldable' clauses. We are going to optimize this predicate 'csub'.

Rule 2. Unfolding

Let C be a clause in P_{i-1}, A a goal in its body and $C_1,\ldots,C_n$ be all the clauses in P_{i-1} whose heads are unifiable with A. Let C_i' $(1 \leq i \leq n)$ be the result of resolving C with C_i upon A. Then let P_i be $(P_{i-1} - \{C\}) \cup \{C_1',\ldots,C_n'\}$ and D_i be D_{i-1}. Mark each C_i' 'foldable' unless it is already in P_{i-1}.

Example (continued)

We unfold C4 at its first goal to obtain $P_2 = \{\underline{C1},\underline{C2},\underline{C3},\underline{C5},\underline{C6},\underline{C7}\}$, $D_2 = \{C4\}$ where the clauses C5,C6 and C7 are listed below.

C5. csub([],Y,Z) ← subseq([],Z)

C6. csub([A|X],[A|Y],Z) ← subseq(X,Y),subseq([A|X],Z)

C7. csub(X,[A|Y],Z) ← subseq(X,Y), subseq(X,Z)

Then C5 is unfolded into

C5'. csub([],Y,Z)

and we get $P_3 = \{\underline{C1},\underline{C2},\underline{C3},\underline{C5}',\underline{C6},\underline{C7}\}$ and $D_3 = \{C4\}$.

The folding rule in our system is not just the inverse of the unfolding rule as it is in the Burstall and Darlington's system. To fold a goal set into a goal, we allow only a clause in D_{i-1} to be used as the folder.

Rule 3. folding

Let C be a clause in P_{i-1} of the form $A \leftarrow A_1,\ldots,A_n$ and C_1 be a clause in D_{i-1} of the form $B \leftarrow B_1,\ldots,B_m$. Suppose there is a substitution θ and a subset $\{A_{i_1},\ldots,A_{i_m}\}$ of the body of C such that the following conditions hold.

1. $A_{i_j} = B_j\theta$ for $j = 1,\ldots,m$,
2. θ substitutes distinct variables for the internal variables of C_1, and moreover those variables do not occur in A or $\{A_1,\ldots,A_n\} - \{A_{i_1},\ldots,A_{i_m}\}$, and
3. C is marked 'foldable' or $m < n$.

Then let P_i be $(P_{i-1} - \{C\}) \cup \{C'\}$ and D_i be D_{i-1} where C' is a clause with head A and body $(\{A_1,\ldots,A_n\} - \{A_{i_1},\ldots,A_{i_m}\}) \cup \{B\theta\}$. Let C' inherit the mark of C.

Example (continued)

Folding the whole body of C7 by C4, we obtain $P_4 = \{\underline{C1},\underline{C2},\underline{C3},\underline{C5}',\underline{C6},\underline{C8}\}$ and $D_4 = \{C4\}$ where C8 is

C8. csub(X,[A|Y],Z) ← csub(X,Y,Z).

To see the need for the condition 2, suppose we fold the clause p(X) ← q(X,Y),r(Y) using a definition s(U) ← q(U,V) into the clause p(X) ← s(X),r(Y). Then the equivalence is destroyed because

the result clause would correspond to a clause p(X) ← q(X,Y1),r(Y) but not to the original one.

The condition 3 prevents for example immediate folding of a definition by itself. Without the condition we fold C4, in P_0 of our example, by itself to end in P_1' = {C1,C2,C3,C4'} where C4' is

csub(X,Y,Z) ← csub(X,Y,Z).

To complete our example, we need one more new predicate.

Example (continued)

Motivated by the failure to fold the body of C6, we introduce an auxiliary predicate 'csub1' and define

C9. csub1(A,X,Y,Z) ← subseq(X,Y), subseq([A|X],Z)

to obtain P_5 = {C1,C2,C3,C5',C6,C8, C9}, D_5 = {C4,C9}.

By unfolding C9 at its second goal, we get P_6 = {C1,C2,C3,C5',C6, C8,C10,C11} and D_6 = {C4,C9} where C10 and C11 are

C10. csub1(A,X,Y,[A|Z]) ← subseq(X,Y),subseq(X,Z)

C11. csub1(A,X,Y,[B|Z]) ← subseq(X,Y),subseq([A|X],Z).

Folding C6, C10 and C11, we obtain the final result P_9 = {C1,C2,C3, C5',C6',C8,C10',C11'} and D_9 = {C4, C9}. Note that C5',...,C11' listed below define the new predicates independently from P_0 = {C1,C2,C3}.

C5'. csub([],Y,Z)

C6'. csub([A|X],[A|Y],Z) ← csub1(A,X,Y,Z)

C8. csub(X,[A|Y],Z) ← csub(X,Y,Z)

C10'. csub1(A,X,Y,[A|Z]) ← csub(X,Y,Z)

C11'. csub1(A,X,Y,[B|Z]) ← csub1(A,X,Y,Z)

When used for generating common subsequences of two given lists, the final program is far more deterministic than the original one because a selection of an element in the first list is immediately checked against the second one. (Of course we are assuming here the fixed order control under which the original program behaves as a typical generare-and-test program.)

The point is that P_9 is equivalent (in the least model semantics) to $P_0 \cup D_9$ and that this is generally true for any transformation sequence obeying the rules. The rest of this section is devoted to the proof of this fact.

2.2 Correctness of the Basic System

First we characterize the least model semantics by means of proof trees. We assume a fixed Herbrand universe and a fixed set of predicates so that the set of ground goals is fixed.

Definition. proof tree

Let S be a program. A tree T, whose nodes are labelled with ground goals, is called a *proof tree*, or simply a *proof*, in S if the following conditions hold.

1. Let A be the root label of T, $T_1,\ldots,T_n$ $(n \geq 0)$ its immediate subtrees and $A_1,\ldots,A_n$ their root labels. Then $A \leftarrow A_1,\ldots,A_n$ is an instance of some clause C in S.

2. Each immediate subtree T_i ($1 \leq i \leq n$) is a proof in S.

We say that T is a proof of A in S and that A is provable (by T) in S. We also say that the clause C is used at the root of the proof T and that $T_1,..,T_n$ are immediate subproofs of T.

In the following, we often argue by induction on the structure of proofs and omit the base case, which is usually subsumed by the induction step as the special case $n=0$, as in the above definition.

The meaning, $M(S)$, of the program S is now defined as the set of all ground goals provable in S. This $M(S)$ is nothing but the least Herbrand model of S (Van Emden 76).

For a transformation sequence $(P_0,D_0),\ldots,(P_N,D_N)$, we define a sequence $S_0,\ldots,S_N$ called *virtual transformation sequence*, by

$$S_i = P_i \cup (D_N - D_i).$$

In particular $S_0 = P_0 \cup D_N$ and $S_N = P_N$. In the following discussion we will always deal with virtual transformation sequences. This amounts to pretending that the definitions of all new predicates are given at the beginning. The set of definitions D_N will be fixed and referred to as D throughout. Since the definition transformation is an identity transformation in the virtual transformation sequence, it will be ignored.

THEOREM

Let $S_0,\ldots,S_N$ be the transformation sequence. Then $M(S_N) = M(S_0)$.

To prove the theorem we need some definitions.

Definition. rank of a ground goal

Let A be a goal in $M(S_0)$ and $r'(A)$ be the size of the smallest proof of A in S_0. Then $r(A)$, the *rank* of A, is $r'(A)$ if A has an old predicate and $r'(A)-1$ if A has a new predicate.

Definition. rank consistent proof

Let S_i be a program in the transformation sequence. Let T be a proof in S_i, C the clause used at its root, $T_1,\ldots,T_n$ ($n \geq 0$) its immediate subproofs, and A, $A_1,\ldots,A_n$ their root labels. Then T is said to be *rank-consistent* if

1. $r(A) \geq r(A_1)+..+r(A_n)$ with equality holding only when C is not marked 'foldable', and
2. $T_1,..,T_n$ are rank consistent.

Now the proof of the theorem consists of showing that the following invariants hold for each i ($0 \leq i \leq N$).

I1. $M(S_i) = M(S_0)$

I2. For each goal A in $M(S_i)$, there is a rank-consistent proof of A in S_i.

The first invariant I1 trivially

holds for $i=0$. As for I2, for any goal A in $M(S_0)$, the smallest proof of A is obviously rank-consistent. (Remember $S_0 = P_0 \cup D$ and the clauses in P_0 are marked 'foldable' while those in D are not.)

The preservation of the invariants is proved in the three lemmas below.

LEMMA 1

If I1 holds for S_i, then $M(S_{i+1}) \subset M(S_i)$.

Proof.

Let A be a ground goal in $M(S_{i+1})$ and T its proof in S_{i+1}. We construct a proof T' of A in S_i by induction on the structure of T.

Let C be the clause used at the root of T, and $T_1,..,T_n$ $(n \geq 0)$ the immediate subproofs of T. By the induction hypothesis we can construct proofs $T_1',..,T_n'$ in S_{i+1} with each T_j' corresponding to T_j. If C is in S_i we can immediately construct T' from C and the proofs $T_1',..,T_n'$. If C is the result of unfolding, we can construct T' from $T_1',..,T_n'$ using the two clauses in S_i of which C is the resolvent.

Now suppose C is the result of folding. Then for some j $(1 \leq j \leq n)$, the root label A_j of T_j is an instance of the folded goal in the body of C. We assume $j=1$. Because A_1 is provable in S_i by T_1', it is also provable in S_0 by the invariant I1. So there should be a ground instance $A_1 \leftarrow B_1,..,B_m$ of some clause in D such that $B_1,..,B_m$ are provable in P_0. Again by I1, $B_1,..,B_m$ are provable in S_i. Let C' be the clause in S_i of which C is the folded result. Owing to the condition 2 of folding, we can combine the proofs of $B_1,..,B_m$ and proofs $T_2',..,T_n'$ with C' to obtain T', the proof of A in S_i. []

LEMMA 2

If the invariants I1 and I2 hold for S_i, then $M(S_i) \subset M(S_{i+1})$.

Proof.

Let A be a ground goal in $M(S_i)$. Then by the invariant there is a rank-consistent proof T of A in S_i. We construct a proof T' of A in S_{i+1} by induction on the well-founded ordering $>>$ defined on $M(S_0)$ $(=M(S_i))$ as

$A >> B$ iff

$r(A) > r(B)$ or

$r(A) = r(B)$ and A has a new and B has an old predicate.

The base case where $r(A)=1$ and A has an old predicate obviously holds because then A should be a ground instance of some unit clause in P_0 which should be in both S_i and S_{i+1}.

Let C be the clause in S_i used at the top of T, and $T_1,..,T_n$ $(n \geq 0)$ the immediate subproofs of T. By the invariant I2, for each root label A_i of T_i, $A >> A_i$ holds. So by the induction hypothesis there are proofs

$T_1', .., T_n'$ of $A_1, .., A_n$ in S_{i+1}. If C is in S_{i+1} the construction of T' is immediate.

Suppose C is unfolded into $C_1, .., C_m$ in S_{i+1} and assume that the root label A_1 of T_1 is the instance of the goal at which C is unfolded. Let $T_{11}, .. T_{1p}$ be the immediate subproofs of T_1, and $A_{11}, .., A_{1p}$ their root labels. Then again by I2 and the induction hypothesis, there are proofs $T_{11}', .., T_{1p}'$ of $A_{11}, .., A_{1p}$ in S_{i+1}. Combining the proofs $T_{11}', .., T_{1p}', T_2', .., T_n'$ with some C_k $(1 \leq k \leq m)$ we get a proof T' of A in S_{i+1}.

Now suppose C is folded into C' in S_{i+1}. Assume that the root labels $A_1, .., A_k$ of $T_1, .., T_k$ $(k \leq n)$ are the instances of the folded goals in C. Let B be a goal such that $B \leftarrow A_1, .., A_k$ is a ground instance of the clause in D used in the folding. By definition, $r(A_1) + .. + r(A_k) \geq r(B)$. By the condition 3 of folding, either C is marked 'foldable', which means $r(A) > r(A_1) + .. + r(A_k)$, or $k < n$. In either cases, $r(A) > r(B)$ holds. Moreover, by the equivalence of S_i to S_0, B is provable in S_i. Therefore by the induction hypothesis, B has a proof T_B in S_{i+1}. Combining the proofs $T_B, T_{k+1}', .., T_n'$ with the clause C', we obtain the proof T' of A in S_{i+1}.

LEMMA 3

If the invariant I1 and I2 holds for S_i, I2 holds for S_{i+1}.

Proof.

We first note that in the proof of lemma 2, T' is constructed in such a way that it is rank-consistent. Thus every goal in $M(S_i)$ has a rank-consistent proof in S_{i+1}. Because $M(S_{i+1}) \subset M(S_i)$ by lemma 1, I2 holds for S_{i+1}. []

This completes the proof of the theorem.

3 GOAL REPLACEMENT

The unfold/fold system becomes more powerful when combined with goal replacement rules.

3.1 General Principle

Let S be a program and $\exists x B_1 \& .. \& B_n$ be an existentially quantified conjunction of goals without free variables. (By x we represent a vector of variables.) We say the formula is provable in S and write $S \vdash \exists x B_1 \& .. \& B_n$ if there is some ground instantiation θ of x such that every $B_i\theta$ $(1 \leq i \leq n)$ is provable in S.

Now let C be a clause in S of the form

$$A \leftarrow A_1, .., A_k, B_1, .., B_m$$

and C' be a clause (not in S) of the form

$$A \leftarrow A_1, .., A_k, B_1', .., B_n'$$

Let $x[y]$ be variables occurring in $B_1, .., B_m$ $[B_1', .., B_n']$ and not

in A, $A_1,..,A_k$ and $B_1',..,B_n'$ [B_1, $..,B_m$].

Suppose for every ground instantiation θ of A, $A_1,..,A_k$ it holds that

$$S\text{-}\{C\} \mid\!\!- \exists x(B_1\&..\&B_m)\theta$$
$$\text{iff} \quad S\text{-}\{C\} \mid\!\!- \exists y(B_1'\&..\&B_n')\theta.$$

Then we can transform S into $S' = (S\text{-}\{C\}) \cup \{C'\}$.

It is rather obvious that the transformation itself preserves the least model. But when we use this rule within the unfold/fold system, we must be careful so that the second invariant I2 of the transformation process is preserved. Consider the following transformation sequence.

P_0: q(s(X)) ← q(X) (1)
q(0) (2)
r(s(X)) ← r(X) (3)
r(0) (4)
Define.
p1(X,Y) ← q(X),r(Y) (5)
p2(X,Y) ← q(X),r(y) (6)
Unfold q in (5).
p1(0,y) ← r(Y) (7)
p1(s(X),Y) ← q(X),r(Y) (8)
Replace r(Y) by r(s(Y)).
p1(s(X),Y) ← q(X),r(s(y)) (9)
Unfold r in (6).
p2(X,0) ← q(X) (10)
p2(X,s(Y)) ← q(X),r(Y) (11)
Replace q(X) by q(s(X)).
p2(X,s(Y)) ← q(s(X)),r(Y) (12)
Fold (9).
p1(s(X),Y) ← p2(X,s(Y)) (13)
Fold (12).
p2(X,s(Y)) ← p1(s(X),Y) (14)

Though each step of goal replacement is valid by itself, the resulting program contains infinite recursion and is not equivalent to the original one. This is because the goal replacement steps destroyed the invariant I2.

The general condition to preserve the invariant I2 is that for every ground instantiation θ of A, $A_1,..,A_k$,

$$r(\exists x(B_1\&..\&B_m)\theta)$$
$$\geq r(\exists y(B_1'\&..\&B_n')\theta) \quad (*)$$

holds, where by $r(\exists zB_1\&..\&B_n)$ we represent the minimum of $r(B_1\sigma)+..r(B_n\sigma)$ for every ground instantiation σ of z.

Under this condition, a rank-consistent proof in S can be converted into a rank-consistent proof in S'. There are many special cases where this condition unconditionally holds.

3.2 Special Cases

goal deletion

Let C be a clause of the form $A \leftarrow B_1,..,B_n$. If for every ground instantiation θ of the clause, $S\text{-}\{C\} \mid\!\!- (B_1\&..\&B_{n-1})\theta$ implies $S\text{-}\{C\} \mid\!\!- B_n\theta$, then B_n can be deleted.

Considering this as the replacement of $B_1,..,B_n$ by $B_1,..,B_{n-1}$, the condition (*) is obviously satisfied.

goal merging

We can merge identical goals in a body into one goal. The condition (*) is also satisfied.

function merging

Suppose there are two goals $p(t_1,..,t_{n-1},x)$ and $p(t_1,..,t_{n-1},y)$ in the body of the clause. Assume further that a ground goal $p(s_1,..,s_n)$ in $M(S\text{-}\{C\})$ is unique up to $s_1,..,s_{n-1}$. (The relation

denoted by p is actually a function.) Then we can merge the two goals applying the substitution $\{y/z\}$ to the rest of the clause. The condition (*) is satisfied.

goal addition

This is the inverse of goal deletion. The following example shows the utility of this seemingly pessimising transformation.

Example (sorting by permutation and order check)

```
P0: perm([],[])
    perm([A|X],Y) ← perm(X,Z),
                    ins(A,Z,Y)
    ins(A,X,[A|X])
    ins(A,[B|X],[B|Y]) ← ins(A,X,Y)
    ord([])
    ord([A])
    ord([A,B|X]) ← A ≤ B,ord([B|X])
Define.
    sort(X,Y) ← perm(X,Y),ord(Y)
Unfold perm.
    sort([],Y) ← ord([])
    sort([A|X],Y) ←
       perm(X,Z),ins(A,Z,Y),ord(Y)
```

Add ord(Z) in the body because for any ground terms t_1, t_2 and t_3, $P_0 \vdash \mathrm{ins}(t_1,t_2,t_3) \& \mathrm{ord}(t_3)$ implies $P_0 \vdash \mathrm{ord}(t_2)$.

```
    sort([A|X],Y) ← perm(X,Z),
         ord(Z),ins(A,Z,Y),ord(Y)
```

Fold the first two goals.

```
    sort([A|X],Y) ←
       sort(X,Y),ins(A,Z,Y),ord(Y)
```

Thus this technique is a vital step from the O(n!) sorting program to an $O(n^3)$ insertion sort program. To obtain an $O(n^2)$ program, however, we need the idea of context (Wegbreit 76), which is beyond the scope of this paper.

Though goal insertion clearly violates condition (*), the above transformation sequence does preserve equivalence. A technique to get around the difficulty will be presented in section 3.3.

laws of primitives

There are various laws for primitive predicates, such as associativity of the predicate 'append' defined by

```
P_ap:  append([],X,X)
       append([A|X],Y,[A,Z])
         ← append(X,Y,Z).
```

We can prove by induction that

$$P_{ap} \vdash^{\exists X} \mathrm{append}(t_1,t_2,X) \,\&\, \mathrm{append}(X,t_3,t_4)$$

iff

$$P_{ap} \vdash^{\exists Y} \mathrm{append}(t_1,Y,t_4) \,\&\, \mathrm{append}(t_2,t_3,Y)$$

for any ground terms $t_1,..,t_4$. So we can apply the associativity of append in any program incorporating P_{ap}.

The condition (*) holds if we use the associativity in one direction (the left hand side of iff to the right hand side), but does not hold in the other direction.

3.3 Weakening the Condition

We have seen that in many cases the goal replacement rule can be used with the unfold/fold transformation unconditionally. But we have also seen interesting cases where the condition (*) does not hold. For such cases we can weaken the condition (*) into the following, (though at the cost of additional bookkeeping of folding conditions.)

For every such θ as in (*), there is a partition of $\{B_1',..,B_n'\}$ such that for each part

$\{B_1'',..,B_j''\}$ of the partition, $r(\exists x(B_1\&..\&B_m)\theta) \geq r(\exists y(B_1''\&..\&B_j'')\theta)$ holds.

In the sorting example, we replaced the goals {ins(A,Z,Y), ord(Y)} by {ord(Z),ins(A,Z,Y), ord(Y)}. This is now justified because $r(\text{ord}(t_2)) < r(\text{ins}(t_1,t_2,t_3)) + r(\text{ord}(t_3))$ for any ground terms t_1, t_2 and t_3 for which the goals are provable. But we have to put labels on the introduced goals as

$$\text{sort}([A|X],Y) \text{ - } \text{perm}(X,Z), \underset{1.1}{\text{ord}(Z)}, \underset{1.2}{\text{ins}(A,Z,Y)}, \underset{1.2}{\text{ord}(Y)}$$

Inheriting these labels through transformation and prohibiting folding of goals of label 1.1 and of label 1.2 together, we can make the induction in the proof of lemma 2 valid.

To prove the correctness of this technique, the condition 1 in the definition of rank-consistency should be changed:

1'. $r(A) \geq r(A_{i_1}) + .. + r(A_{i_m})$ for any subset $\{A_{i_1},..,A_{i_m}\}$ of $\{A_1,..,A_n\}$ such that no two goals in the subset have imcompatible labels, with equality holding only when C is marked 'foldable'.

The detail of the modified proof is omitted.

Note that the folding of the first two goals in the above clause does not violate the label constraint, so that the whole example of sorting is justified. The reverse direction of the associativity of 'append' can also be handled in this manner.

4 CLAUSE ADDITION/DELETION

clause addition

Let C be a clause not in S. If for every ground instance $A \leftarrow A_1,..,A_n$ of C, $S \mid\!- A_1\&..\&A_n$ implies $S \mid\!- A$, we can add C to S.

clause deletion

Let C be a clause in S. If for every ground instance $A \leftarrow A_1,..,A_n$ of C, $S - \{C\} \mid\!- A_1\&..\&A_n$ implies $S - \{C\} \mid\!- A$, we can delete C from S.

The correctness of these transformations themselves is again obvious. When combined with the unfold/fold transformation, clause addition causes no problem. Clause deletion can in general destroy the invariant I2 of the transformation process. As in the case of goal replacement, there are important special cases.

Let C and C' be clauses in S of the forms $A \leftarrow A_1,..,A_n$ and $B \leftarrow B_1,..,B_m$ such that A is an instance $B\sigma$ of B. Let $x[y]$ be the sequence of variables in $A_1,..,A_n$ $[B_1,..,B_m]$ but not in $A[B]$. If for every ground instantiation θ of A, $S - \{C\} \mid\!- \exists x(A_1\&..\&A_n)\theta$ implies $S - \{C\} \mid\!- \exists y(B_1\&..\&B_m)\sigma\theta$, then C can be deleted. In this special case of goal deletion, the condition

$$r(\exists x(A_1\&..\&A_n)\theta) \geq r(\exists x(B_1\&..\&B_m)\sigma\theta) \text{ for every } \theta$$

guarantees the preservation of the invariant I2. In particular, if $\{B_1\sigma,..,B_m\sigma\} \subset \{A_1,..,A_n\}$,

which means syntactic subsumption, the condition is trivially satisfied.

Finally, it should be remarked that clause addition/deletion, unlike goal replacement, are often used apart from the unfold/fold system. In such cases we need not worry about the invariant.

5 CONCLUDING REMARKS

We have proved the correctness of the basic unfold/fold system and then examined the interaction of the augmenting transformation rules with the correctness property. We have stated a sufficient condition for their application to be safe. To ensure the equivalence of the result of some transformation sequence with the initial program, we need only to check the condition for each application of those rules. As we have seen, in many useful special cases this involves only a simple syntactic checking. In other cases, proving the condition can be a difficult task. However, we can still claim the advantage over the usual separate equivalence proof approach because we have the choice of either keeping the conditions through transformation sequence or proving separately the equivalence of the result with the original program.

Though one might expect that the unfold/fold system preserves stronger properties like completion or finite failure(Clark 1978) (Apt and Van Emden 1982), this is not the case for these properties. There are easy counter examples.

The practical power of the system depends on the heuristics we employ: we have a large search space generated by the choice of applicable transformation rules. We are currently investigating this strategic aspect with some experimental implementations.

ACKNOWLEDGEMENTS

We would like to thank Yoshihiko Futamura for initiating our interest in this area, Hozumi Tanaka, Toshio Yokoi and Kouichi Furukawa for encouragement, Koukichi Futatsugi and other members of ETL for helpful discussions, Rodney Harries for correcting the English of the first manuscript, and Kazuko Hata for typing the final manuscript.

REFERENCES

Apt, K.R. and Van Emden, M.H. Contributions to the theory of logic programming. Journal of the ACM 29, No. 3, 1982.

Burstall, R.M. and Darlington, J. A transformation system for developing recursive programs. Journal of the ACM 24, No. 1, 1977.

Clark, K.L. and Sickel, S. Predicate logic: a calculas for deriving programs. Procs. IJCAI-77, Boston, 1977.

Clark, K.L. Negation as Failure, in H. Gallaire and J. Minker(eds), Logic and Databases, Plenum Press, 1978.

Clark, K.L. Predicate logic as a computational formalism. Imperial College research monograph 79/59 TOC, December 1979.

Hogger, C.J. Derivation of logic programs. Journal of the ACM 23, No. 4, 1976.

Kowalski, R.A. Predicate logic as a programming language. IFIP 74, North Holland Publishing Co., 1974.

Tamaki, H. and Sato, T. A transformation system for logic programs which preserves equivalence. ICOT TR-018, July 1983.

Van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming languages. Journal of the ACM 23, No. 4, 1976.

Wegbreit, B. Goal-directed program transformation. 3rd POPL symposium, Tucson, January 1976.

BOUNDED-HORIZON SUCCESS-COMPLETE RESTRICTION OF INFERENCE PROGRAMS

Michel Sintzoff

Unité d'Informatique
Université Catholique de Louvain
chemin du Cyclotron 2
B-1348 Louvain-la-Neuve, Belgium

ABSTRACT

The paper deals with the control of search in logic programming (Horn clause inference) by the addition of restrictive predicates to rules so as to cut off all blind-alleys without loosing possible results. Criteria are proposed to ensure that additional premises in clauses allow to establish results without trials-and-errors. These criteria require neither the introduction of special well-orderings nor the induction of limits of predicates. They take into account structural properties of bounded-length compositions of the original clauses, and consequently are only sufficient.

1. INTRODUCTION

Exhaustive search, or backtracking, remains a fundamental stumbling-block on the way to economic use of programs based on inference techniques, such as canonical Post productions, definite Horn clauses, elementary formal systems (Smullyan 1961), or van Wijngaarden two-level grammars. To overcome that obstacle, four lines of work have been explored, and continue to be so. Firstly, various non-von-Neumann computer architectures are designed in order to permit high concurrency in the parallel exploration of alternative computation paths. Secondly, refined evaluation regimes are introduced so as to abandon unsuccessful paths earlier, thanks to an analysis of previous failures; typical techniques are the alpha-beta heuristics and the intelligent backtracking (Bruynooghe 1978). Thirdly, lower-level statements are made available for the explicit programming of more efficient control-flow and data-flow; simple illustrations are the sequentialization of clauses and the cut-operator in Prolog. Fourthly, inference programs can be specialized by adding in clauses selective premises which eliminate blind alleys; this is known as the "logic control by logic" (Pereira 1982) and can be seen as an instance of efficiency improvement by program transformations; successful and important examples of this approach in other formal systems are the generation of parsers for context-free languages and the Knuth-Bendix technique.

The work reported here focuses on the derivation of deterministic programs from nondeterministic specifications by bounded-depth transformations. Thus it follows the fourth, transformational method. A major problem proves to be the lack of elementary means for ensuring the adequacy of additional, selective premises. On the one hand, these should remove all deadends. On the other hand, they

should not remove too much, viz. they should preserve the possibility of deriving each result computable by the original program. As a rule, in order to deduce selective premises correctly, or simply to validate these, termination functions are introduced. But then the problem is to discover appropriate orderings; this is typically the case for methods based on the Knuth-Bendix completion procedure (Dershowitz 1982). The present work explores an alternative technique, namely the use of straightforward criteria for ensuring the adequacy of selective premises; whenever these criteria are verified and yield deterministic programs, it is guaranteed that all blind-alleys are cut off and that no possible result is lost. These criteria directly use structural properties of the original inference clauses, including their commutativity, equivalence, or idempotence. They do not require the introduction of intelligent orderings, the perspicacious induction of exact limits of iterations on predicates, or the construction of insightful proofs in logic. However, these criteria are only sufficient, not necessary: they take into account bounded-length compositions of inference steps. For this reason, they are called "bounded-horizon".

The first part of the paper introduces "inference programs" which are Horn clauses (van Emden and Kowalski 1976) organized hierarchically and expressed in a canonical form. Auxiliary concepts are introduced, a.o. the success domain of an inference program and the inclusion relation between clauses. The proposed criteria are then developed in terms of linear recursive clauses: this simplifies the technical developments and corresponds to the way the criteria have been found out. The generalization to non-linear inference programs is introduced afterwards.

Only the validity of the proposed criteria is proved in detail since they constitute the main contribution. The other, subservient results should be clear enough, and are merely presented. References (Shoenfield 1967, van Emden and Kowalski 1976, Dijkstra 1976) provide a useful background for predicate calculus, Horn clauses, and weakest preconditions, respectively.

Notations:

A <-B1,...,Bn : Horn clause
v, &, =>, ~ : or, and, then, not
?A : a thesis predicate

T.i stands for T3 when i=3
$i:T.i stands for T1 + T2 + ...

2. INFERENCE PROGRAMS

An inference program T comprises a basic clause T0 and a set Tr of inductive clauses T.i (i>0); the identifier Tr suggests "the recursive part of T":

$$\begin{aligned} T &= T0 + Tr \\ Tr &= \$i:T.i \\ &= T1 + T2 + \ldots + Tn \end{aligned} \qquad (2.1)$$

The basis clause T0 has the form

$$p(x) \leftarrow p0(x) \qquad (2.2)$$

where p is the principal predicate symbol, and p0 is the basis predicate symbol. Each inductive clause T.i has the form

$$p(x) \leftarrow r.i(x,y),\ p(y) \qquad (2.3)$$

where r.i denotes a known relation between the tuples x and y.

A top-down computation consists

in an iterative application of modus tollens using the inductive and basis clauses. A basis computation step is

```
?p(z)        p(x) <- p0(x)
-------------------------        (2.4)
          ?p0(z)
```

An inductive computation step is

```
   ?p(z)
   p(x) <- r.i(x,y), p(y)
  ------------------------       (2.5)
   ∃w: r.i(z,w) &  ?p(w)
```

The preimage of a predicate Q(x) through an inductive clause T.i is

```
 T.i(Q)                          (2.6)
    = ∃y: r.i(x,y) & Q(y)
```

For Tr=($i:T.i), the preimage is

```
 Tr(Q) = T1(Q) v...v Tn(Q)       (2.7)
```

The "success domain" or relation Rel(T) defined by an inductive program T = T0 + Tr is the set of tuples which can be proved to verify p by using (2.4-5):

```
 Rel(T) = least solution         (2.8)
          of [X = p0 v Tr(X)]
```

As usual, this least solution can be computed iteratively:

```
 Rel(T) = ∃n: H.n      (n≥0)

where       H0 = p0               (2.9)
      H.(n+1) = p0 v Tr(H.n)
              = H.n v Tr(H.n)
```

The formulae (2.8-9) are valid because Tr is continuous, hence monotonic; indeed,

```
 Tr(∃n:Q.n)
  = ∃i:∃y: [r.i(x,y) & ∃n:Q.n(y)]
  = ∃n:∃i:∃y: r.i(x,y) & Q.n(y)
  = ∃n: Tr(Q.n)
```

In particular,

```
 Tr(Q1 v Q2)
  = Tr(Q1) v Tr(Q2)              (2.10)
```

Note that, in general, we only have

```
 Tr(Q1 & Q2)
  => Tr(Q1) & Tr(Q2)             (2.11)
```

The reverse implication holds provided each r.i(x,y) is functional w.r.t. x. In this case,

```
 T1(Q1) & T1(Q2)
  =  ∃y,z: r1(x,y) & Q1(y)
         & r1(x,z) & Q2(z)
  => ∃y: r1(x,y) & Q1(y) & Q2(y)
  => T1(Q1 & Q2)
```

The relation r.i(x,y) in an inductive clause (2.3) may embody syntactical as well as semantical constraints, corresponding to unifications and to auxiliary premises respectively. A thesis takes the form

```
   ∃x=(x',x"): (x'=e) & ?p(x)
```

where the expression e is known; this means that x' and x" are the known and unknown parts of the tuple x, respectively. A top-down computation "succeeds" if it rewrites the thesis into a predicate which does not contain the principal predicate symbol p anymore and which yields true thanks to successful proofs of all the subtheses depending on auxiliary predicates p0 and r.i. Given a successful computation, the conjunction of the definitions of the successive new variables such as w in (2.5), corresponds to an "answer substitution" or a "recursion stack". Top-down computations as defined here reduce conjunctions of positive atoms to true; this is isomorphic to the reduction to false of disjunctions of negative atoms, as is customary

with the resolution principle.

Example of inference program:

An append program can be written as follows; x and y are triples, and app((x1, x2, x3)) stands for "to append x1 to x2 yields x3":

```
T0: app(x) <- p0(x)
T1: app(x) <- r1(x,y), app(y)
```

The auxiliary predicates p0 and r1 are defined by

```
p0((x1, nil, x1)) <- true
r1((x1, a.x2, a.x3), (x1, x2, x3))
                  <- true
```

Consider then the thesis

```
∃x: (x1=1.nil) & (x2=2.nil)
    & ?app(x)
```

By (2.5), this is rewritten as

```
∃x: (x1=1.nil) & (x2=2.nil)
    & [∃w: r1(x,w) & ?app(w)]
```

which can be simplified to

```
∃x,w: (x3=2.w3)
      & ?app((1.nil, nil, w3))
```

By (2.4), the latter becomes

```
∃x3,w3: (x3=2.w3)
        & ?p0((1.nil, nil, w3))
```

The last atom is true for w3=1.nil. Thus we proved

```
∃x: (x1=1.nil) & (x2=2.nil)
    & (x3=2.1.nil) & app(x)
```

viz.

```
app((1.nil, 2.nil, 2.1.nil)).
```

3. RESTRICTION OF PROGRAMS

To restrict an inference program amounts to add premises in clauses. The restriction S.i of an inference clause T.i by a predicate C.i is such that

$$\forall Q: S.i(Q) = C.i \;\&\; T.i(Q) \qquad (3.1)$$

Namely,

```
  T.i: p(x) <-  r.i(x,y), p(y)
  S.i: p(x) <- r'.i(x,y), p(y)
```

for r'.i(x,y) = C.i(x) & r.i(x,y).

The restriction S of an inference program T by predicates C.i's is the result of restricting each inductive clause T.i by C.i, according to (3.1). Clearly,

$$\forall Q: Sr(Q) \Rightarrow Tr(Q) \qquad (3.2)$$

where Sr = ($i:S.i).

Hence, by (2.9), Rel(S) => Rel(T). Accordingly, the restricted program S does not permit the successful computation of a thesis for which no computation by T succeeds.

The restriction of T into S is success-complete if Rel(S) equals Rel(T): no tuple valid for T is lost by S.

The restricted program S is deterministic if the selective premises C.i are mutually exclusive and depend only on known values in all computation steps:

```
  C.i & C.j   = false, for i≠j,
and, for all i,
  C.i & p0    = false
  C.i(x', x") = C'.i(x')
```

where x' and x" respectively represent the known and unknown parts of the tuple x. Thus, in any computation step, at most one clause can be applied successfully, and the choice of that clause can be made using the available information, without delay.

If the restricted program S is deterministic and success-complete w.r.t. T, then the top-down computations by S from all the theses

computable by T do succeed necessarily, without trials-and-errors. Indeed, because of completeness, for each thesis acceptable by T there must exist a successful top-down computation by S: the property

$$\forall a', a^*: [(a', a^*) \in Rel(T) \Rightarrow (a', a^*) \in Rel(S)]$$

entails

$$\forall a': [\exists a^*: (a', a^*) \in Rel(T)] \Rightarrow [\exists a^*: (a', a^*) \in Rel(S)]]$$

Moreover, determinism prevents backtracking: since there must exist a successful path for each acceptable thesis, and since there may exist at most one path, no blind-alley may be entered.

Consequently, it is most useful to transform an inference program T into a deterministic, success-complete equivalent. This is possible under the following assumption, using the notations hereabove: for all valid input x', there exist a unique output x* such that (x',x*) belongs to Rel(T). Otherwise, we should weaken as follows the definition of success-completeness: for each input x', if Rel(T) contains some (x,x*), then Rel(S) must contain some (x,x'''); alternatively, we could relax the constraint of determinism.

The remainder of the paper develops sufficient criteria for the success-completeness of given restrictive premises.

4. INCLUSION RELATION

We propose a simple way of telling when clauses are "included in" other ones. A couple of auxiliary properties are first introduced. Using (x=z) for Q(x) in (2.6) yields

$$T.i(x=z) = \exists y: r.i(x,y) \;\&\; (y=z) = r.i(x,z) \qquad (4.1)$$

By (4.1), definitions (2.6) and (2.7) entail

$$T.i(Q) = \exists y: T.i(x=y) \;\&\; Q(y)$$
$$Tr(Q) = \exists y: Tr(x=y) \;\&\; Q(y) \qquad (4.2)$$

Similarly, in general,

$$Tr(Sr(Q)) = \exists y: Tr(Sr(x=y)) \;\&\; Q(y) \qquad (4.3)$$

The <u>inclusion relation</u> between two inductive clauses T1 and T2 is defined by

$$\forall Q: T1(Q) \Rightarrow T2(Q) \qquad (4.4b)$$

This is entailed by

$$\forall y: T1(x=y) \Rightarrow T2(x=y) \qquad (4.4a)$$

Indeed, by (4.2,4.4a),

$$T1(Q) = \exists y: T1(x=y) \;\&\; Q(y) \Rightarrow \exists y: T2(x=y) \;\&\; Q(y) \Rightarrow T2(Q)$$

Thanks to (4.4a), the inclusion relation between clauses can be checked without considering all possible predicates Q.

This can be generalized as follows; Tr is a set of inductive clauses; T.j, T.i, T.k are inductive clauses; the range of k is left understood:

$$\text{IF} \quad \forall y: [\, T.j(T.i(x=y)) \Rightarrow (x=y) \vee Tr(x=y) \vee \exists k: T.k(Tr(x=y)) \,] \qquad (4.5a)$$

$$\text{THEN} \quad \forall Q: [\, T.j(T.i(Q)) \Rightarrow Q \vee Tr(Q) \vee \exists k: T.k(Tr(Q)) \,] \qquad (4.5b)$$

Indeed, by (4.3,4.5a,4.2),

$$T.j\,(T.i(Q)) = \exists y: T.j(T.i(x=y)) \;\&\; Q(y)$$

```
   => ∃y: [(x=y) v Tr(x=y)
           v ∃k:T.k(Tr(x=y))] & Q(y)
   => Q v Tr(Q) v ∃k: T.k(Tr(Q))
```

5. BOUNDED-HORIZON CRITERIA

Given (2.9), Rel(T) = Rel(S) is implied by the following, for n>0:

$$\forall n: H.n = K.n \qquad (5.1)$$

where
$$H1 = Tr(p0) \qquad (5.2)$$
$$K1 = Sr(p0)$$
$$H.(n+1) = H.n \vee Tr(H.n)$$
$$K.(n+1) = K.n \vee Sr(K.n)$$

We look for simple criteria which merely ensure the existence of a proof of (5.1), without requiring to construct such a proof actually. The idea is to compile the induction, that is to abstract useful structural properties from families of actual proofs by induction, and to use these structural properties as criteria. To start with, we consider the set of proofs by induction with depth two: it is a typical case, neither too trivial nor too elaborated. The corresponding criteria are called "bounded-horizon criteria of rank two", and are noted BH[2]. Program S is a restriction (3.1) of T; p0 is the basis predicate (2.2); I is the index set of the inductive clauses (2.3).

Bounded-horizon criteria BH[2]:

$$Tr(p0) \Rightarrow Sr(p0) \qquad (5.3)$$

$$Tr(Sr(p0)) \Rightarrow Tr(p0) \vee Sr(Tr(p0)) \qquad (5.4)$$

$$\forall J \subseteq I: \exists j \in J: \forall i \in I:$$

$$[\,[\,T.j(C.i) \Rightarrow Tr(p0) \vee Tr(Tr(p0)) \vee \exists j \in J: C.j\,] \vee \forall y:[\,T.j(T.i(x=y)) \Rightarrow (x=y) \vee Tr(x=y) \vee \exists k \in I \setminus J: T.k(Tr(x=y))]] \qquad (5.5)$$

The criteria (5.3) and (5.4) ensure the two basis steps, and (5.5) guarantees the induction step, for an induction of depth two.

Theorem. Let the inference program S be the restriction of a program T by selective premises C.i's. If the criteria (5.3-5) are verified, then S is success-complete w.r.t. T.

Proof. Let us show that the criteria (5.3-5) do ensure that a proof of (5.1) by induction on n exists.

Base n=1: (5.3) expresses H1=>K1, and (3.2) entails K1=>H1.

Base n=2: Given H1=K1, we may rewrite (5.4) as H1 v Tr(H1) => K1 v Sr(K1). This expresses H2=>K2; (3.2) entails K2=>H2.

Induction step n>1: the induction hypotheses are

$$H.(n-1) = K.(n-1) \qquad (5.6)$$
$$H.n = K.n$$

The induction thesis is

$$H.(n+1) = K.(n+1)$$

By (5.2,5.6,3.2), this induction thesis is reduced to

$$Tr(K.n) \Rightarrow H2 \vee H.n \vee Sr(H.n)$$

We unfold K.n and H.n by (5.2):

$$Tr(K.(n-1)) \vee Tr(Sr(K.(n-1))) \Rightarrow H2 \vee H.(n-1) \vee Tr(H.(n-1)) \vee Sr(H.(n-1)) \vee Sr(Tr(H.(n-1)))$$

We observe $Sr(H.(n-1)) \Rightarrow Tr(H.(n-1))$ because of (3.2), and use (5.6) again; the induction thesis becomes

$$Tr(Sr(H.(n-1)) \qquad (5.7)$$

```
   => H2 v H.(n-1) v Tr(H.(n-1))
      v Sr(Tr(H.(n-1)))
```

On the other hand, (5.2) and (5.6) imply the following property:

```
H.(n-1) v Tr(H.(n-1))          (5.8)
  = H.(n-1) v Sr(H.(n-1))
```

We thus have to show the existence of a proof of (5.7) on the basis of (5.8), viz. a proof of (5.8) |- (5.7). We do it for a version obtained by abstracting from H.(n-1), viz. by substituting an indeterminate predicate Q for H.(n-1):

```
∀Q: Q v Tr(Q) = Q v Sr(Q)      (5.9)
  |- Tr(Sr(Q))
   => H2 v Q v Tr(Q) v Sr(Tr(Q))
```

It remains to prove (5.9) on the basis of the criterion (5.5). We first note that the antecedent in (5.9) entails T.k(QvTr(Q)) = T.k(QvSr(Q)) for any k ranging over some subset of I, say I\J. Hence, applying (2.10) and adding ∃j∈J:T.j(Q) on both sides, we may assume

```
Tr(Q) v ∃k:T.k(Tr(Q))          (5.10)
   = Tr(Q) v ∃k:T.k(Sr(Q))
```

Let us recall the identity

```
(∃i∈I: D.i) = ∃J⊆I: (∀j∈J: D.j)
                  & (∀k∈I\J: ~D.k)
e.g.(D1vD2) = (D1&~D2) v (D2&~D1)
              v (D1&D2)
```

We use this identity for the term Tr(Sr(Q)) = [∃i∈I: Ti(Sr(Q))]. The consequent of (5.9) then becomes

```
   [∃J⊆I:    (∀j∈J:  D.j)
         & (∀k∈I\J: ~D.k)]
=> H2 v Q v Tr(Q) v Sr(Tr(Q))
```

where each D.h stands for T.h(Sr(Q)). We move the quantifier on J to the front of the whole implication and the one on k within the conclusion, and we use (5.10) for weakening Sr into Tr within the k-terms:

```
∀J⊆I: [∀j∈J: T.j(Sr(Q))          (5.11)
       => H2 v Q v Tr(Q)
          v Sr(Tr(Q))
          v ∃k∈I\J: T.k(Tr(Q))]
```

We may write Sr(Tr(Q)) as

```
  Sr(Tr(Q))
     = ∃j∈J:   C.j & T.j(Tr(Q))
     v ∃k∈I\J: C.k & T.k(Tr(Q))
```

Each j-term in Sr(Tr(Q)) can be simplified to C.j because the antecedent of (5.11) contains each T.j(Sr(Q)), and because Sr(Q) => Tr(Q) by (3.2). Each k-term in Sr(Tr(Q)) can be removed altogether since it is included in the k-terms already present in the conclusion of (5.11). Thus the thesis (5.11) can be written as follows, after moving the quantifier on j to the front:

```
∀J⊆I: ∃j∈J:
[ T.j(Sr(Q))
   => H2 v Q v Tr(Q) v ∃j∈J:C.j
        v ∃k∈I\J: T.k(Tr(Q))  ]
```

We unfold Sr(Q) into [∃i∈I: C.i & T.i(Q)] and use T.j(C.i & T.i(Q)) => T.j(C.i) & T.j(T.i(Q)), given (2.11); a thesis A=>C can be replaced by a thesis B=>C when A=>B. We move the quantifier on i to the front and apply the identity (A&B => CvD) <=> (A=>C) v (B=>D); hence the new thesis:

```
∀J⊆I: ∃j∈J: ∀i∈I:                (5.12)
  [T.j(C.i) => H2 v ∃j∈J:C.j]
v [T.j(T.i(Q))
          => Q v Tr(Q)
             v ∃k∈I\J: T.k(Tr(Q))]
```

Now, by assumption, the criterion (5.5) is verified; because of (4.5), it entails

```
∀J⊆I: ∃j∈J: ∀i∈I:                    (5.13)
[ [T.j(C.i) => H2 v ∃j∈J: C.j]
v ∀Q: [T.j(T.i(Q))
         => Q v Tr(Q)
           v ∃k∈I\J: T.k(Tr(Q))]]
```

Formula (5.13) does imply (5.12), because [A v ∀Q:B] => ∀Q:[A v B] and ∃j∀Q:Z => ∀Q∃j:Z. QED

Example of use of the criteria:

The greatest common divisor f(x1,x2) can be defined by

```
f(x1,x1) = x1, f(x1,x2) = f(x2,x1)
     f(x1,x1+x2) = f(x1,x2)
```

The relation gcd((x1,x2,x3)) <=> (f(x1,x2)=x3) can thus be defined by

```
T0: gcd(x) <- p0(x)
T1: gcd(x) <- r1(x,y), gcd(y)
T2: gcd(x) <- r2(x,y), gcd(y)
```

plus the auxiliary clauses

```
p0((x1,x1,x1))      <- true
r1((x1,x2,x3),(x2,x1,x3))
                    <- true
r2((x1,x2,x3),(x1,y2,x3))
                    <- (x2=x1+y2)
```

The program T0,T1,T2, permits infinite computation paths because of the permutating clause T1. In order to eliminate the unsuccessful paths without loosing solutions, it is proposed (aha!) to restrict T1 and T2 by using the two selective premises

```
C1 = (x1>x2),   C2 = (x2>x1)
```

The program thereby obtained is deterministic. To show it is success-complete w.r.t. T, we apply the criteria BH[2]. The verification of (5.3) and (5.4) is immediate. For (5.5), we sketch the central steps.

The case J={1}, i=1, holds because T1(T1(x=y)) = (x=y); T1 is idempotent, and thus it is of no use to apply T1 twice. The case J={1}, i=2, holds because T1 transforms C2 into C1, viz. T1(C2) => C1: the domain from which T1 establishes C2 is contained in C1.

For J={2}, i=1, we must check

```
    [ T2(C1) => H2 v C2 ]
v ∀y: [ T2(T1(x=y))
          => (x=y) v Tr(x=y)
            v T1(Tr(x=y))  ]
```

where Tr=T1+T2. The part T2(C1)=>C2 amounts to (x1+x1>x2 => x2>x1), i.e. (x1<x2). The term T2(T1(x=y)) amounts to (x2-x1, x1, x3)=y which is false when x1 ≥ x2. Hence we do have [x1<x2] v ∀y: [T2(T1(x=y)) => ...].

For J={2}, i=2, we verify T2(C2) => C2 viz. (x2>x1+x1 => x2>x1).

Finally, all the cases J={1,2}, for any j and i, yield true because H2 v C1 v C2 is identically true: indeed, H2 contains x2=x1 since p0 implies x1=x2 and T1 exchanges x1 and x2.

This simple example illustrates how the proposed criteria (5.3-5) directly benefit from structural properties of the original inference clauses: the more such properties are available, the less constraints on the selective premises must be verified. This is to be contrasted with the Knuth-Bendix approach, where properties such as idempotence, commutativity, associativity, require additional work, viz. special well-orderings.

6. GENERAL INFERENCE PROGRAMS

The adaptation of the previous developments to the non-linear case is straightforward. We begin with the quadratic case. The in-

ductive clause T.i (2.3) takes the form

```
p(x) <- r.i(x,y,y'),        (6.1)
        p(y), p(y')
```

The inductive computation step (2.5) becomes

```
  ?p(z)              T.i      (6.2)
-------------------------------
?∃w,w': r.i(z,w,w') & p(w)&p(w')
```

The preimage of two predicates Q and Q' is

```
T.i(Q,Q')                    (6.3)
  = ∃y,y': r.i(x,y,y')
          & Q(y) & Q(y')
```

Accordingly, (2.7-9) respectively become

```
Tr(Q,Q') = ∃i: T.i(Q,Q')     (6.4)

Rel(T) = least solution      (6.5)
         of [X = p0 v Tr(X,X)]

       = ∃n: H.n             (6.6)
where     H0 = p0
     H.(n+1) = H.n v Tr(H.n, H.n)
```

The generalizations of (2.10) and (2.11) are clear, viz.

```
Tr(Q1 v Q2, Q1' v Q2')
  = Tr(Q1,Q1') v Tr(Q1,Q2')
  v Tr(Q2,Q1') v Tr(Q2,Q2')
```

Similarly,

```
T.i(x=z, x=z')
  = ∃y,y': r.i(x,y,y')
         & (y=z) & (y'=z')
  = r.i(x,z,z')              (6.7)
```

The restriction (3.1) of an inductive clause is now defined using

```
r'.i(x,y,y')                 (6.8)
    = C.i(x) & r.i(x,y,y')
```

The inclusion relation between non-linear clauses has two forms. In the first case, the predicates Q, Q' used as parameters are independent:

```
IF   ∀y,y':                  (6.9a)
     T1(x=y,x=y') => T2(x=y,x=y')

THEN ∀Q,Q':                  (6.9b)
       T1(Q,Q')  => T2(Q,Q')
```

Indeed, by (6.7,6.9a,6.3),

```
T1(Q,Q')
 =  ∃y,y': T1(x=y, x=y')
          & Q(y) & Q'(y')
 => ∃y,y': T2(x=y, x=y')
          & Q(y) & Q'(y')
 => T2(Q,Q')
```

In the second case, the same predicate Q is used for both parameters:

```
IF ∀y,y':                    (6.10a)
   T1(x=y, x=y')
   => T2(x=y ,x=y) v T2(x=y ,x=y')
    v T2(x=y',x=y) v T2(x=y',x=y')

THEN ∀Q:                     (6.10b)
       T1(Q,Q) => T2(Q,Q)
```

Indeed, by (6.7,6.10a, 6.3),

```
T1(Q,Q)
 = ∃y,y':
    T1(x=y, x=y') & Q(y) & Q(y')
 => ∃y,y',z,z':
   (z=y v z=y') & (z'=y v z'=y')
  & T2(x=z, x=z') & Q(y) & Q(y')
 => ∃z,z':
    T2(x=z, x=z') & Q(z) & Q(z')
 => T2(Q,Q)
```

The case (6.10) with identical parameters is the one used hereafter, because of the form of the equations in (6.5-6). We shall abbreviate (6.10a) into

```
∀y,y': ∃z,z' ⊆ y,y':
  T1(x=y, x=y') => T2(x=z, x=z')
```

The inclusion property (4.5) is generalized similarly:

```
IF  ∀y1...y4:                                (6.11a)
    ∃z1...z4 ⊆ y1...y4:
    [ T.j(T.i1(x=y1, x=y2),
          T.i2(x=y3, x=y4))
      => (x=z1) v Tr(x=z1, x=z2)
        v ∃k: T.k(Tr(x=z1, x=z2),
                  Tr(x=z3, x=z4)) ]

THEN ∀Q:                                     (6.11b)
     [T.j(T.i1(Q,Q), T.i2(Q,Q))
     => Q v Tr(Q,Q)
      v ∃k: T.k(Tr(Q,Q),Tr(Q,Q)) ]
```

The bounded-horizon criteria of rank two for the quadratic case have the same structure as (5.3-5): one merely replaces the linear compositions by their quadratic generalizations. The proof of the theorem of section 5 is obtained similarly: one systematically replaces Tr(Y) by Tr(Y,Y) and Sr(Y) by Sr(Y,Y), for any Y and as long as possible; for instance, Sr(Tr(H.(n-1))) becomes

```
Sr(Tr(H.(n-1), H.(n-1)),
   Tr(H.(n-1), H.(n-1)))
```

One can check that the steps of the transformed proofs are validated by the use of (6.1) to (6.11). In order to derive the transformed version of (5.12), the following fact is used in addition; for any T1, T2, T3, C1, C2, Q,

```
T3(C1&T1(Q,Q), C2&T2(Q,Q))
  => T3(C1,C2)
     & T3(T1(Q,Q), T2(Q,Q))
```

The difference between the linear case in section 5 and the quadratic case essentially amounts to the difference between linear compositions of computation steps and quadratic compositions of steps in a binary tree of computations.

The above generalization can be extended to inference programs of any degree in homogeneous form: the binary trees become n-ary. A program is in homogeneous form of degree n if all its inductive clauses have exactly n premises which use the principal predicate symbol. Such an homogeneous form can always be obtained by adding redundant premises in clauses. For instance, p(y) is equivalent to p(y) & p(y') & (y=y').

7. DISCUSSION

The research on methods of transforming formal systems for the complete elimination of backtracking without any loss of solutions, is as practically important as technically hard. Note this only tackles a restricted problem, in comparison to more general methods for the formal derivation of efficient inference programs; these may demand "genuinely deep theorems requiring mathematically challenging proofs" (Hogger 1981).

Two major classes of transformation methods for backtrack-elimination are well established. The first one concerns the generation of deterministic parsers for context-free production systems, which correspond to a very restricted form of inference programs. The methods in the second class transform equational systems into confluent systems of rewriting rules (Knuth and Bendix 1970); since equational systems are comparable in power and structure to inference programs, it is possible to use these methods for Horn clauses also (Barbuti, Degano and Levi 1982). The fundamental difference between the Knuth-Bendix approach and the method explored here is that the latter does not depend on the definition of ad-hoc well-orderings and actually benefits from properties such as idempotence or commutativity in the original program clauses; but this does not imply that the present method is any better in

practice, especially in view of systems such as REVE (Lescanne 1983).

The first results of the present approach appeared in (Sintzoff 1976): ideas based on bounded-context parsing methods were developed for computations by "transition" programs or by inference ones. Transition programs are sets of condition-action pairs, where boolean expressions and substitutions respectively stand for conditions and actions; they correspond to restricted guarded-command loops, without nesting. The inference programs were expressed by two-level grammars similar to Horn clauses. The criteria in (Sintzoff 1976) prove quite limited, and apply if the initially given programs permit at most one successful computation path for each input; this restriction originates from the fact that parsing methods assume the context-free grammars are not ambiguous. In (van Lamsweerde and Sintzoff 1979), parallel programs are studied in the form of transition programs, and specialization techniques are given to eliminate deadlock and starvation; these techniques demand the actual induction of greatest and least fixpoints, and thus are general but difficult to use. A first version of bounded-horizon criteria is developed in (Sintzoff 1978), for transition programs only; a weaker variant is studied in (Sintzoff 1983) to prevent failures after finite paths, but without cutting off infinite blind-alleys.

The present work pursues these investigations and applies the results to inference programs. Much more work clearly is needed. Fully constructive ways of using the criteria BH[2] must be developed; to use bounded-horizon criteria for proving or improving programs, even mechanically, appears promising: any set of selective premises C.i's solving (5.3-5) is acceptable, not just extremal solutions. Generalizations and variants should be investigated, e.g. by allowing for bounded horizons of rank greater than two, by permitting nondeterministic selective premises, by not giving priority to shorter compositions of computation steps over longer ones, or by ensuring specific data-flows in the clauses. Last but not least, the practical applicability of the proposed technique must be explored.

To the cost of appearing foolish and presumptuous, we venture some personal views on the relationship between the present work and artificial intelligence. See (Rich 1983) for a presentation of the relevant technical concepts.

Instead of weak search implementations, we look for strong ones, e.g. transformations which yield absolutely deadend-free production systems. In particular, we want to derive conditional and iterative plans for parametrized goals and by use of an homogeneous design strategy.

Incompleteness is inescapable in a bounded world. Here, it concerns only the bounded-horizon transformation process: this one may well fail but whenever it succeeds, it yields a program which is guaranteed to be success-complete. The transformation is weak whereas the result, if any, is strong. It is at the program design level that the spirit of heuristics reigns.

The term "bounded-horizon" refers also to the burning issue of the horizon effect. We try to tackle this problem "statically",

by analyzing specific composition properties of rules, before searching w.r.t. singular goal states. To use bounded-horizon criteria of rank two for "remote horizons" may seem as naïve as approximating pi by 22/7, but could prove as attractive as the LR[2] parsing techniques for context-free grammars; who knows?

Naïve systems, built on basic knowledge and inference compilation, must complement expert systems made of knowledge bases and inference engines.

REFERENCES

Barbuti, R., P. Degano, and G. Levi, Towards an inductionless technique for proving properties of logic programs, in: Proc. 1st Intern. Logic Progr. Conf., Marseille, 1982, 175-181.

Bruynooghe, M., Intelligent backtracking for an interpreter for Horn clause logic programs, Report CW16, Univ. Leuven, Belgium, 1978.

Dershowitz, N., Orderings for term-rewriting systems, Theor. Computer Sci. 17(1982), 279-301.

Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs NJ, 1976.

Hogger, C.J., Derivation of logic programs, J. ACM 28(1981), 372-392.

Knuth, D., and P. Bendix, Simple word problems in universal algebras, in: Computational Problems in Abstract Algebra, Pergamon, London, 1970, 263-297.

Lescanne, P., Computer experiments with the REVE term rewriting system generator, in: Proc. 10th Conf. on Principles of Progr. Languages, ACM, 1983, 99-108.

Pereira, L.M., Logic control with logic, in: Proc. 1st Intern. Logic Progr. Conf., Marseille, 1982, 9-18.

Rich, E., Artificial Intelligence, McGraw-Hill, Tokyo, 1983.

Shoenfield, J.R., Mathematical Logic, Addison-Wesley, London, 1967.

Sintzoff, M., Eliminating blind alleys from backtrack programs, in: Proc. 3rd Int. Coll. Automata, Languages and Programming, Edinburgh Univ. Press, 1976, 531-557.

Sintzoff, M., Ensuring correctness by arbitrary postfixed-points, in: Proc. 7th Symp. Math. Found. Comp. Sci., LNCS 64, Springer, Berlin, 1978, 484-492.

Sintzoff, M., Issues in the methodical design of concurrent programs, in: A.W. Biermann and G. Guiho (eds.) Computer Program Synthesis Methodologies, D. Reidel, Dordrecht, 1983, 51-78.

Smullyan, R.M., Theory of Formal Systems, Princeton Univ. Press, 1961.

van Emden, M.H., and R.A. Kowalski, The semantics of predicate logic as a programming language, J. ACM 23(1976), 733-742.

van Lamsweerde, A., and M. Sintzoff, Formal derivation of strongly correct concurrent programs, Acta Informatica 12(1979), 1-31.

AN EFFICIENT BUG LOCATION ALGORITHM

David A. Plaisted
Department of Computer Science
University of Illinois
1304 West Springfield Avenue
Urbana, Illinois 61801 USA

ABSTRACT

We present an efficient algorithm for locating bugs in Prolog. This algorithm is based on the method of (Shapiro 1983), and can be applied to any high level programming language. The method is optimal to within a constant factor for space, time, and number of queries to the user. This significantly improves the performance of Shapiro's method, which is not optimal for space or time and for which the number of queries depends on the branching factor of the computation. Since no current programming environment uses this method, it should be a significant aid to programmers in debugging software.

1. INTRODUCTION

Probably millions of dollars of computer time are spent each day by programmers tracing their programs to locate errors and understand their programs better. This often requires repeated execution of parts of the program in order to locate a bug. Typically the programmer will execute a top-level procedure to find which subprocedure returns incorrect values; he or she will then execute the subprocedure to find which of its subprocedures return incorrect values; and this process continues until the error is found. This process can be quite time consuming for programs with large execution times, so much so that much of this tracing is probably not done because of the excessive cost, and other methods are used to debug programs.

Ideally, programs should probably be written using some kind of program transformation scheme or program verifier to help insure their correctness; however, in practice, program testing and debugging is the main programming methodology used. Current programming environments such as INTERLISP permit a programmer to examine the stack when a run time error occurs. This is often not sufficient because the error may not be in a procedure invocation currently on the stack. Another alternative is to insert trace statements in the program. This is also not satisfactory for long executions, since there may be thousands of lines of output to examine.

It would be a significant aid to

This research was supported in part by the National Science Foundation under grants MCS-81-09831 and MCS-83-07755.

program debugging if a more efficient method of searching computation trees were available. This would save not only programmer time and computer time, but would also make practical certain kinds of tracing that are currently prohibitive in cost. It turns out that there is such an efficient method for searching computation trees; the basic idea is that the results of selected subcomputations are remembered so that those subcomputations need not be repeated. These subcomputations are carefully chosen and the manner of examining the tree is carefully structured so that the increase in computation time to search the tree can be made arbitrarily small. The method may be viewed as a generalization of binary search to trees.

Shapiro has given one method (Shapiro 1983) of searching the computation tree. Our method is similar to his "divide and query" method but has the following advantages: 1. The number of queries does not depend on the branching factor. This can be significant if the branching factor is large, say, a thousand, which is conceivable in a program with iterative statements, if not in Prolog(Clocksin and Mellish 1981). 2. The storage required is bounded by a constant factor of that required by the original program. This is not true of Shapiro's method, in which the storage required may be a factor of log(t) times that required by the original execution, where t is the execution time. However, many Prolog implementations save all intermediate results until the end anyway, so this factor often is not important for Prolog programs. 3. The increased execution time for searching the computation tree can be made arbitrarily small in our method. That is, if the original program execution took say an hour, then it will be possible to find the bug in an extra 5 minutes of computation time using our method, with an appropriate number of intermediate results saved. However, in Shapiro's method, it may be necessary to run the whole program again, to learn its execution time, and then run it a third time to find the bug. The third execution may take as long as the first, so the total execution time can be increased by a factor of 3. It is not difficult to modify Shapiro's method to eliminate the second execution to determine the execution time, but even with such a modification, there is still a factor of 2 in execution time required.

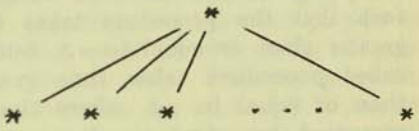
Our method may be used for Prolog or for other high-level programming languages, but there may be difficulties for languages with pointers. Also, if there are arrays, the storage requirement can be large. Shapiro also mentions a "top down" query strategy which requires little extra storage, but which may square the execution time. This "top down" method can be made more time efficient, at the expense of a possibly large amount of storage. We do not consider the problem of nontermination; this may be approached by related methods, as Shapiro mentions (Shapiro 1983). Another approach to oracles is given in (Edman and Tarnlund 1983) where methods for semi-automating the construction of an oracle are given. They are concerned with the problem of guaranteeing the correctness of the oracle, and show how a correct oracle may be constructed from program specifications.

We may consider the program execution as a tree, where a procedure invocation P that calls subprocedures $P_1, \cdots, P_k$ corresponds to a node in the tree labeled P with sons labeled $P_1, \cdots, P_k$. The object is to find a procedure invocation such that the procedure P returned a wrong result but all subprocedures executed correctly; this procedure then contains a bug. To find this erroneous invocation, queries are given to the user asking if a procedure invocation with specified input and output is correct. If the program as a whole contains a bug, one can show (Shapiro 1983) that there must be some procedure invocation that is erroneous in the above sense. Our method is essentially a fast method of examining computations of programs, with little overhead in storage or time; it appears that no current system (such as INTERLISP) contains a comparable method.

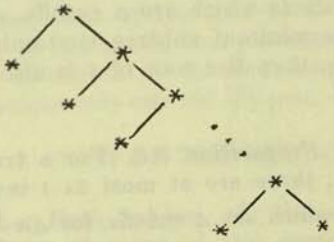
2. REDUCING THE BRANCHING FACTOR

We first give a method for transforming the execution tree to reduce the branching factor to two. The *branching factor* is the maximum number of sons of any node in the tree; the method in (Shapiro 1983) is sensitive to the branching factor. Suppose a procedure invocation P calls procedures $P_1, \cdots, P_k$. We allow queries of the form, "If procedure P was called with such and such inputs, then should it be possible to reach a state after P_j returns in which the variables accessible to P have such and such values?" Thus we can determine if an error has occured before the end of the j^{th} procedure call, using a single query.

Note that if $j=k$ then this is equivalent to asking if P itself returned with correct values. (For languages having global variables, these must also be included in the values used and returned by P since P may use and change them.) This has the effect of transforming a subtree of the form



to a subtree of the form



3. CHOOSING I/O PAIRS

The method works as follows: A program is run. During the run of this program, certain procedure invocations are chosen and the inputs and outputs to these are stored. If the program terminates with a correct answer then nothing need be done. Suppose the program terminates with a wrong answer. Then some of the selected i/o pairs are used to query the user about the correctness of the corresponding procedure invocations. Based on the results of these queries,

either the incorrect invocation is found or else a smaller subcomputation is found which contains the bug; the method is then applied recursively to this subcomputation. Note that no recomputation is necessary until all relevant queries based on stored i/o pairs have been made.

The i/o pairs are chosen as follows: A procedure invocation is called a Δ *cutoff* if there exists an integer n such that the procedure takes time greater than or equal to $n\Delta$, but no called procedure takes time greater than or equal to $n\Delta$, where time is measured by numbers of procedure calls. One can show the following:

Proposition 3.1. For any Δ, the Δ cutoffs form an upper semilattice. That is, if p and q are two invocations which are Δ cutoffs, and s is the minimal subtree containing p and q, then the root of s is also a Δ cutoff.

Proposition 3.2. For a tree of size t, there are at most $2n-1$ invocations which are Δ cutoffs, for $\Delta \geq \frac{t}{n}$.

Proposition 3.3. Suppose a procedure invocation p takes t_p time units, and $t \geq 2\alpha$, and $\alpha \geq \Delta$. Then there is an invocation q in the subtree of p such that q is a Δ cutoff and q takes time t_q for $\alpha \leq t_q \leq 2\alpha$. (We are assuming the branching factor is two, as above.)

There is a problem because initially we may not know how long the program is going to run. We have to store enough procedure calls so that whenever the program stops, the stored invocations will form a set of Δ cutoffs for small enough Δ. We show how to do this in the next section.

3.1. Storing i/o pairs

Suppose that at some time, all δ cutoffs have been stored, and δ is about $\frac{t}{n}$, where the execution time is t. Then only the 2δ cutoffs are saved, and only 2δ cutoffs are stored until the execution time is about $2t$. This process is then repeated (i. e. only the 4δ cutoffs are stored until time about $4t$) until the execution ends. At any time, Δ cutoffs are stored for $\frac{t}{n} \leq \Delta \leq 2\frac{t}{n}$. This requires the storing of at most $2n-1$ i/o pairs, by proposition 3.2. During the execution, it is necessary to keep a counter with each procedure call telling the time at which it occurred, so that when the procedure returns we may know how long it took and whether it is a Δ cutoff. Also, it is necessary to save the inputs to each procedure until it exits. In languages such as Prolog, this is no problem, since the backtracking mechanism requires that this information be saved. In other languages, such values may need to be explicitly saved, costing possibly some extra storage. If the procedure changes global variables, their original values will also have to be saved.

4. QUERYING THE USER

Suppose all Δ cutoffs have been stored, for $\Delta \leq \frac{t}{n}$. This may require storing $4n-1$ i/o pairs during the execution of the program, by above reasoning. Then the user is queried by a kind of "binary search". The first query is some Δ cutoff p such that the

time t_1 taken by p is between $\frac{t}{3}$ and $\frac{2t}{3}$, where t is the execution time of the program. Such a cutoff p exists by proposition 3.3. If p is correct, then a subtree of size at least $\frac{t}{3}$ is known not to contain the bug; if p is incorrect, a subtree of size at most $\frac{2t}{3}$ is known to contain the bug. In either case, at least $\frac{1}{3}$ of the tree is eliminated from consideration by this query.

Such queries are continued on the relevant subtree, possibly with certain of its subtrees known to be correct. In a constant times log(n) queries, the user will have located a cutoff q which is incorrect but such that all cutoffs in the subtree rooted at q are correct. In fact, by proposition 3.1, there will be at most two maximal cutoffs q_1 and q_2 below q which are known to be correct. Therefore this series of queries has reduced the size of the region in which the bug may occur to at most $2\frac{t}{n}$. The method is then continued on this reduced region, with q_1 and q_2 considered to be eliminated from the subtree. This is possible since their i/o pairs have been stored, and any recomputations need not repeat the computations below q_1 or q_2.

5. ELIMINATING OLD I/O PAIRS

The above method can be costly in storage, since the q_i need to be stored, and they can accumulate during each pass of the method. To get around this problem, we give a method that insures that the number of such old i/o pairs is never more than two. The idea is to save extra i/o pairs during the next pass of the method, which when queried will narrow the search for the bug down to a subtree in which at most one such old i/o pair exists.

For example, if there are two such old i/o pairs r_1 and r_2, let T be the minimal tree containing both of them. Then during the next pass of the algorithm, we save the i/o pair for the root of T, as well as the i/o pairs for its two sons. Then we query the user whether the root of T is correct. If so, the entire subtree can be eliminated and we only have to store the i/o pair for the root of T instead of both r_i. Suppose the root is incorrect. Then we query for the sons. If both are correct, then the bug has been found; it is at the root of T. If one son is incorrect, then attention may be restricted to that subtree, which contains only one old i/o pair.

6. COST OF THE METHOD

Each phase of the method reduces the size of the computation tree from t to $\frac{2t}{n}$. Therefore the $\frac{2t}{n}$ part of the computation must be repeated, and so on, leading to a computation time of $t+\frac{2t}{n}+\frac{4t}{n^2}+\cdots$ or $\frac{n}{n-2}t$. For n large enough, the extra computation time can be made very small.

The number of i/o pairs that must be stored is $4n-1$, since extra pairs need to be stored to account for the fact that the total execution time of the program may not be known in advance. Also, possibly 3 i/o pairs

may be needed for the extra queries to eliminate the buildup of old i/o pairs, and at most one old i/o pair needs to be stored. Thus we may need $4n+3$ i/o pairs. The number of queries to the user will be $O(\log t)$. For the method to work, n must be at least 3, so 15 i/o pairs may be needed. Probably this can be reduced significantly; if arrays need to be kept with each i/o pair, even 15 pairs may be excessive.

7. BACKTRACKING

In the above discussion we have assumed that no backtracking occurs. If backtracking does occur, the method can be modified to deal with it. A procedure call may fail, in which case the calling procedure must try to find an alternative procedure call or sequence of calls, or else must itself fail. Each state in the sequence of calls may be considered a possible query, of the form "If procedure P is called with such and such input values, is it possible to reach a state in which $P1, \cdots, Pj$ have returned successfully and the values of the variables accessible to P are such and such?" Note that calls which fail may lead to bugs, since possibly they should have succeeded.

8. VARIATIONS

If the desired i/o relation of a procedure P is given procedurally, then instead of querying the user, it is possible to query some program which tests whether P returns correct values. In this case, our methods are still useful if the testing program requires a large amount of execution time. For example, if we are testing to see if a fast, complicated program is equivalent to a slow, simple program, the slow program may be used as an automatic query answerer for the fast program, but it is best not to query the slow program very often. Shapiro has observed that answers to previous queries can be remembered, to further reduce the number of queries.

Another variation of the method is to query the user during the first execution of the program, even before the user knows that it contains a bug. That is, as each i/o pair as in section 3.1 is stored, it is also used as a query to the user. It is easy to show that the total number of queries is increased at most by a constant factor by this method. However, if the program is unreliable, then it probably will give wrong results early in the computation, and this can be detected early. Also, if each i/o pair is used as a query as soon as it is stored, then fewer i/o pairs need to be stored. For example, suppose p and q are i/o pairs which are stored, and q is in the subtree of p (that is, q is an invocation of a procedure during the computation of p). If p is correct, then it is not necessary to save q. Thus fewer pairs need to be saved. On the average this reduces the number of i/o pairs stored at any given time to be proportional to the log of the number required otherwise.

REFERENCES

Clocksin, W. and Mellish, C., Programming in PROLOG, Springer-Verlag, Berlin, 1981.

Edman, A. and Tarnlund, S., Mechanization of an oracle in a debugging system, Proceedings of IJCAI-83,

Karlsruhe, Germany, 1983, 553-555.

Shapiro, E., Algorithmic Program Debugging, MIT Press, Cambridge, Massachusetts, 1983.

OR-PARALLELISM ON APPLICATIVE ARCHITECTURES[1]

Gary Lindstrom[2]
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, MA 02139

ABSTRACT

In a previous paper we introduced an abstract model for OR-parallel logic program execution, oriented toward applicative architectures. Central to this method is pipelined processing of streams of substitution data objects. We now address two implementation issues associated with this approach:

1. The efficient representation of substitution data objects, and

2. A parallel unification algorithm compatible with this representation.

Our approach to the first issue involves a compact vectorized representation permitting indexed access of local variable bindings. Results on the second issue exploit a formulation of unification as a *write once* database update problem, which can be efficiently implemented by a particular combination of applicative and imperative architectural features.

[1]This research was supported in part by National Science Foundation Grant MCS 7915255.

[2]Currently on sabbatical leave from the University of Utah.

1. BACKGROUND

1.1 OR- vs. AND-Parallelism

Efforts to exploit parallelism in the execution of logic programs may be categorized into two domains (Conery and Kibler 1981):

OR-parallelism, where multiple clauses unifiable with a goal literal are attempted concurrently, and

AND-parallelism, where multiple literals within a clause body are attempted concurrently.

OR-parallelism is implementationally simpler, since the alternative clauses under consideration are logically independent (Haridi and Ciepielewski 1983, Furukawa et al. 1982, Umeyama and Tamura 1982, Warren 1984). However, control of eagerness in OR-parallelism is known to be a problem.

In contrast, the concurrent goals pursued under AND-parallelism are not logically independent, since they are generally 'cooperatively' seeking to bind one or more shared variables. This cooperation provides a basis for concurrency control, if read/write disciplines are placed on the shared variable occurrences within a clause. Some semantic and implementational complexities are incurred, but the net effect is a more familiar 'process oriented' view of the computation. Two principal approaches to AND-parallelism through shared variable control have appeared (Clark and Gregory 1983, Shapiro 1983).

We believe that a blend of OR- and AND-parallelism will prove most effective in the long run, but that the best such combination will depend on the underlying architecture employed. For this reason, we are investigating parallel logic programming on a particular applicative architecture named *Rediflow* (Keller et al. 1984), with the strategy of first understanding the implications of OR-parallelism. AND-parallelism will be subsequently introduced, when our implementation understanding has grown and (perhaps) a consensus has arisen at large on what forms of AND parallelism are most desirable.

1.2 Issues in Implementing OR-Parallelism

Given that we wish to develop an OR-parallel logic programming implementation on an applicative architecture, two general issues arise:

1. *Multiple environments:* Under sequential logic programming implementations, e.g. Warren's Prolog compiler for the DECSystem-10 (Warren 1977), there is only one binding environment in existence at a time. The others are 'hidden', and are restored as necessary under backtracking. For this reason, variable binding by destructive writes into unique locations can be utilized (assuming that references to the changed variables are retained, e.g. on a 'trail list', so that the bindings can be later undone if necessary). The result is a 'shallow binding' effect, similar to that used in interpretive Lisp systems, but with the simplification that values to be restored are uniformly the pseudo-value *unbound*, which we denote by the atom Ω, representing the lack of a binding.

In contrast, several instances of a given goal can be concurrently active under OR-parallelism. Hence multiple logical environments must exist logically, though complete physical separation is potentially costly.

2. *Parallel unification:* A unification algorithm is needed which:

   a) is compatible with the multiple environment requirement (i.e. will bind variables such that they are 'shared' within a clause instance, but 'non-shared' among OR-siblings of that instance), and

   b) exploits well the potential concurrency in typical unification invocations.

In a previous paper (Lindstrom and Panagaden 1984) we presented a model for an OR-parallel execution method based on compositions of substitution data objects. Since that paper's completion, we have refined the method to deal with the two important problems just cited. We now describe these refinements.

2. REVIEW OF BASIC APPROACH

2.1 Key Features

Our previously reported approach is based on the following ideas:

1. a stream-based analog of the 'standard' backtracking execution model (in particular, left-to-right pursuit of goals within clause bodies);

2. OR-parallelism, with a particular form of induced AND-parallelism (eager passing of subsolutions to AND-siblings);

3. an applicative formulation, except for indeterminate stream merging (we will weaken this a bit further);

4. concurrent processing of several top-level goals, if desired, and

5. a 'pure code' utilization of program clauses, with all instantiation done via composition of substitution records.

2.2 Binding Operations

Central to our method is the use of substitution objects as the sole means of representing environments. For a detailed treatment of the associated mechanics, the

reader is directed to the previous paper. However, we summarize here the essential ideas necessary to understand the issues of immediate interest.

Initially, we will assume the substitutions are represented symbolically (i.e. as sets of assignments on identifiers). The major operations on substitutions are the following:

1. *Goal instantiation:* Let $\mathcal{T}_0$ be a goal literal appearing in a clause C_0, and $\mathcal{S}_1$ be a substitution on $\mathcal{V}(C_0)$, the set of variables appearing in C_0 (the 'native' name space of C_0). Then $\mathcal{T}_0' = \mathcal{T}_0 \circ \mathcal{S}_1$ denotes the instantiation of this goal by $\mathcal{S}_1$ (a 'goal instance'). We term $\mathcal{S}_1$ an *instantiation* substitution, and stipulate that $range(\mathcal{S}_1) \subseteq domain(\mathcal{S}_1)$, where $domain(\mathcal{S})$ (resp. *range*) is the set of variables appearing on the *left* (resp. *right*) of assignments in a substitution $\mathcal{S}$. Thus instantiation substitutions:

a) have domain and range in the same name space, and

b) permit the important effect of binding *cascading*, whereby chains of bindings are established (e.g. X := f(a, Y), Y := g(Z, b), etc.). This possibility, as we shall see, is a natural consequence of unification, where bindings can be incrementally refined.

2. *Subgoal unification:* Now suppose we wish to attempt unification of $\mathcal{T}_0'$ with literal $\mathcal{T}_1$, the head of a *target clause* C_1. We represent the success of this unification by the substitution pair $[\mathcal{S}_2, \mathcal{S}_3]$, such that $\mathcal{T}_0' \circ \mathcal{S}_2 = \mathcal{T}_1 \circ \mathcal{S}_3$. We term $\mathcal{S}_2$ a *unification substitution*; $\mathcal{S}_3$ is of course an instantiation substitution, playing the same role for $\mathcal{T}_1$ that $\mathcal{S}_1$ does for $\mathcal{T}_0$.

As a special condition, we require that $\mathcal{S}_2$ map any unbound variables in $\mathcal{T}_0'$ into terms in the name space of C_1 (possibly augmented to accomplish this). Thus we insist that $range(\mathcal{S}_2) \subseteq domain(\mathcal{S}_3)$. There is no cascading of bindings in unification substitutions.

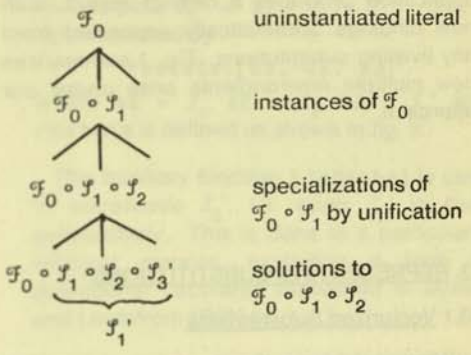


Fig. 1. Multiple environments under substitution application.

3. *Solution restatement:* By our method, each solution to $\mathcal{T}_1 \circ \mathcal{S}_3$ will be represented as a refinement $\mathcal{S}_3'$ of $\mathcal{S}_3$, so $domain(\mathcal{S}_3) \subseteq domain(\mathcal{S}_3')$. (The domain of $\mathcal{S}_3'$ may include an expansion to accommodate new unbound variables contained in the solution of $\mathcal{T}_1 \circ \mathcal{S}_3$; this will be clarified later.) Moreover, since $range(\mathcal{S}_2) \subseteq domain(\mathcal{S}_3)$, the solution conveyed by $\mathcal{S}_3'$ may be *restated* in terms of $\mathcal{V}(C_0)$ by the composition $\mathcal{S}_1' = \mathcal{S}_1 \circ \mathcal{S}_2 \circ \mathcal{S}_3'$. $\mathcal{S}_1'$ is then used to instantiate the right AND-sibling of $\mathcal{T}_0$, or, if $\mathcal{T}_0$ is the rightmost literal in its clause, for solution restatement in terms of the name space of the parent goal of $\mathcal{T}_0$.

The data structures presented in our previous account provide a mechanism for matching a solution substitution (i.e. $\mathcal{S}_3'$) with its associated goal instance, as remapped by the associated unification substitution (i.e. $\mathcal{T}_0' \circ \mathcal{S}_2$). This is done by packaging instantiation substitutions within chained 'application objects', the details of which are inconsequential here.

It is important to note that the functional nature of this technique (i.e. its reliance solely on substitution application) ensures the environment separation effects required

under OR-parallelism. Each substitution application produces a distinct object, with new bindings automatically separated from any existing substitutions. Fig. 1 summarizes how multiple environments arise under our approach.

3. REPRESENTING SUBSTITUTIONS

3.1 Vectorized Substitutions

We now address the need for a 'compiled' substitution representation. The symbolic representation we have used heretofore has two significant drawbacks:

1. binding lookup is associative, i.e. by identifier keys, and

2. awkward name conflict problems can arise upon variable importation.

We adopt the following efficiency criteria for our new representation:

1. direct access of bindings without searching;

2. control of physical copying costs, and

3. compactness, whereby an instantiation substitution $\mathcal{I}_1$ should have a physical size on the order of $|domain(\mathcal{I}_1)|$.

Our solution is of course to use a vectorized representation, with local variables compiled into serial indices. We will denote such indices as 'V1', 'V2', ... meaning 'the local variable with index 1', '... index 2', etc. Our vectorized substitutions will include two portions:

an *initial* portion, equal in length to the number of local variables in the clause involved, and

an *extension*, required to represent bindings of variables 'imported' by instantiation.

Sample clause:

p(X, Y) :- q(X, f(Z)), r(Z, Y, c).

Compiled form:

p(V1, V2) :- q(V1, f(V3)), r(V3, V2, c).

Sample instantiating substitution (symbolic):

[X := b, Y := g(a, W), Z := Z, W := W]

Sample instantiating substitution (vectorized):

b	g(a, V4)	Ω	Ω
V1 (X)	V2 (Y)	V3 (Z)	V4 (W)

Fig. 2. Vectorized substitution representation.

The variable importation effect is a special requirement of our technique for representing environments by substitutions. We require that *every* unbound variable appearing in a goal instance be mapped by unification onto a term in the target clause name space. This may be seen in fig. 2, for example, where the variable W is introduced into the clause's name space. Hence *every* variable as yet unbound in a goal instantiation has a *local* instance in the resulting substitution. This is the essense of our technique for representing environments by substitution data objects.

3.2 Applying Substitutions

We now indicate how substitutions can efficiently be applied to literals using our selected representation. There are two occasions where such applications occur.

During unification: A unification attempt involves an instantiated goal literal (e.g. $\mathcal{I}_0 \circ \mathcal{I}_1$) and the uninstantiated head literal (e.g. $\mathcal{I}_1$) of a target clause. If successful, this produces a substitution pair $[\mathcal{I}_2, \mathcal{I}_3]$, as

```
FUNCTION apply2|[s1, s2]|t =
  IF   isvar:t
  THEN IF   eq:[tselect:[t, s1],
                unbound]
       THEN tselect:[t, s2]
       ELSE apply2|[s1, s2]|
              (tselect:[t, s1])
  ELSE IF   atom:t
       THEN t
       ELSE apply2|[s1, s2] || t,

FUNCTION restate:[s1, s2, s3] =
  {s3b = bindsubst:s3,

  FUNCTION f:i =
      IF   eq:[tselect:[i, s1],
                unbound]
      THEN tselect:[
             tselect:[i, s2],
             s3b]
      ELSE tselect:[i, s1]],

  RESULT make:['tuple,
             tlength:s1, f]},

FUNCTION bindsubst:s =
  {sb = bindterm || s,

  FUNCTION bindterm:t =
     IF   isvar:t
     THEN tselect:[t, sb]
     ELSE IF   atom:t
          THEN t
          ELSE bindterm || t,

  RESULT sb}
```

Fig. 3. Substitution application functions (`isvar` detects variable occurrences).

discussed previously. In section 4.1 we will see that $\mathcal{F}_2$ is actually constructed prior to the application of $\mathcal{F}_1$ to $\mathcal{F}_0$, so the image of $\mathcal{F}_0$ used during unification is $\mathcal{F}_0 \circ \mathcal{F}_1 \circ \mathcal{F}_2$. This is represented by

```
apply2|[s1, s2] || f0
```

where `s1` = $\mathcal{F}_1$, `s2` = $\mathcal{F}_2$, `f0` = $\mathcal{F}_0$ in tuple representation, and `apply2` is defined as shown in fig. 3.

During solution restatement: Here we wish to compute $\mathcal{F}_1' = \mathcal{F}_1 \circ \mathcal{F}_2 \circ \mathcal{F}_3'$. This is accomplished by

```
restate:[s1, s2, s3]
```

where `s1` = $\mathcal{F}_1$, `s2` = $\mathcal{F}_2$, `s3` = $\mathcal{F}_3'$, and `restate` is defined as shown in fig. 3.

The auxiliary function `bindsubst` is used to *decascade* $\mathcal{F}_3'$, i.e. apply $\mathcal{F}_3'$ to itself exhaustively. This is done in a particularly efficient manner, exploiting a form of definitional circularity discussed in (Keller and Lindstrom 1981).

The functions in fig. 3 are expressed in the Function Equation Language FEL (Keller 1982), which resembles ISWIM (Burge 1975) in many ways. The following comments should help clarify the notation:

Block expressions are denoted

{*equations* **RESULT** *expression*}

where the *equations* define locally bound names, generally used within the result expression. Static scoping rules apply. In addition,

1. Right associative function application is denoted by an infix colon, i.e. `f:x:y` = `f:(x:y)`.

2. "`|`" denotes left associative function 'Currying,' i.e. `f|x|y` = `(f:x):y`. When used in a function heading, e.g.

```
apply2|[s1, s2]|t
```

a Curried or 'multi-tiered' function is defined. Hence `apply2` may be invoked in expressions of the form

```
(apply2:[s1, s2]):t,
```

or simply `apply2|[s1, s2]|t`.

3. "`||`" in FEL denotes tuplewise application ('apply-to-all'), i.e. `f||[x1, ..., xk]` = `[f:x1, ..., f:xk]`.

4. Selection of the i-th component of a tuple `t` is denoted `tselect:[i, t]`; `head:t` offers a synonym for `tselect:[1, t]`. The length of a tuple `t` is obtained by `tlength:t`.

5. In addition to direct creation by 'displays' of the form `[v1, ..., vk]`, tuples may be created indirectly by via the utility function `make:['tuple, k, f]`, which yields the tuple `[f:1, f:2, ..., f:k]`. Tuples of writeable cells may be created via `makecells`.

3.3 A Special Problem

There is a flaw in the `restate` function given in fig. 3, in the case where solutions contain unbound variables. An example is the unit clause C:

contemporaries(fatherof(V1), motherof(V1))

unified with the goal instance

contemporaries(V1, V2)

under the instantiating substitution $\mathcal{S}_1$ = [Ω, Ω]. Unification produces $[\mathcal{S}_2, \mathcal{S}_3]$ =

[[V2, V3], [Ω, fatherof(V1), motherof(V1)]].

The resulting $\mathcal{S}_3'$ = $\mathcal{S}_3$, and by the `restate` function defined above we obtain the malformed substitution

$\mathcal{S}_1'$ = [fatherof(Ω), motherof(Ω)]

due to accesses of the unbound variable V1 in $\mathcal{S}_3'$. Under our vectorized approach, treating this effect correctly requires relocating the indexed representation for V1 and extending $\mathcal{S}_1'$ one component position. That is, we should instead obtain

$\mathcal{S}_1'$ = [fatherof(V3), motherof(V3), Ω].

Since this difficulty is comparable to the variable importation problem during unification, we defer discussion of correcting `restate` until section 4.5, after our unification technique is presented.

4. PARALLEL UNIFICATION

We now consider the issue of efficient unification within this framework. From our standpoint, there are four important aspects of the problem:

1. recognition of concurrency potential within the task;

2. exploitation of that concurrency through straightforward use of applicative implementation techniques;

3. appropriate synchronization controls to ensure consistent binding of shared variables, and

4. compatibility with our vectorized approach.

Unification has of course been intensely studied as a sequential algorithm; the recent algorithm of (Martelli and Montanari 1982) is representative of the current state of the art. Indeed, recent results indicate that in certain pathological cases, unification is inherently sequential in nature (Dwork et al. 1983). However, it is clear that in typical unification applications considerable potential for parallelism can arise nevertheless. For example, when variable occurrences are unique within the terms to be unified, concurrency on the order of the arity of the terms is clearly possible.

Our approach will seek to exploit such typically available concurrency, while observing necessary synchronization controls when multiple bindings of a given variable are attempted. This will be achieved by:

viewing unification as a special 'write once' database update problem, and by

utilizing a particular combination of applicative and imperative language features.

Note our objective here is the smooth integration of unification into our overall evaluation method, in which concurrency arises primarily through OR-parallelism. By exploiting whatever concurrency is available (albeit limited) within each unification attempt, two benefits result on an architecture such as Rediflow:

1. greater activity breadth (i.e. 'enabled instructions') within each processing element (PE), thereby reducing the chance of PE idleness due to memory latency, and

2. speedier determination of failing unification attempts.

4.1 Unification as a Database Problem

To begin, we simplify our unification problem to a more familiar form in which bindings are collected in a single substitution, rather than in the $[\mathcal{I}_2, \mathcal{I}_3]$ pair suggested above. We accomplish this by constructing $\mathcal{I}_2$ prior to actual commencement of the unification algorithm, as follows. Suppose $\mathcal{T}_0' = \mathcal{T}_0 \circ \mathcal{I}_1$ is to be unified with $\mathcal{T}_1$, the head of a clause C_1. Let n be $|\mathcal{V}(C_1)|$. Then for $\mathcal{I}_2$ we create a (nonwriteable) tuple with length($\mathcal{I}_2$) = length($\mathcal{I}_1$), as follows:

If $\mathcal{I}_1[i] = \Omega$, then $\mathcal{I}_2[i] = V(n + UB(i, \mathcal{I}_1))$, where $UB(i, \mathcal{I}) = |\{k \lceil k \leq i \text{ and } \mathcal{I}[k] = \Omega\}|$.

Otherwise, the value of $\mathcal{I}_2[i]$ is undefined, and no accesses will be made through it.

The result is a mapping of $\mathcal{T}_0'$ by $\mathcal{I}_2$ into the name space of C_1, as extended by the importation of images of all unbound variables in $\mathcal{T}_n'$. Since $\mathcal{I}_2$ simply serves a variable 'relocation' function, it can be fixed at unification set up time. Then all bindings during the actual unification process are done via assignments to $\mathcal{I}_3$.

Now, let us consider parallel unification as a 'classical' database update problem within an applicative framework, e.g. as formulated in (Keller and Lindstrom 1982). Here:

The 'database' is the vector $\mathcal{I}_3$, initialized to uniformly Ω values.

The database system consists of a stream of *transactions* applied to a stream of *database versions*. Each transaction involves an indivisible *access and bind* operation, which reads a variable's binding, and, if equal to Ω, binds it. The response generated for the transaction indicates whether the binding was adopted, or, if not, what binding is already in effect for that variable.

The execution of each transaction yields in addition an updated database version ($\mathcal{I}_3$ vector), which is then fed back in a cyclic fashion, to be paired with the next transaction arriving. An overall transaction serialization effect thereby results.

This approach is quite clean functionally, relying on a single pseudo functional operator, *viz.* the stream merge used to collect transactions for application against the database. However, from a pragmatic viewpoint, this approach is suboptimal, for the following reasons:

1. All accesses of the database are made mutually exclusively, when in fact serialization on a per-variable basis is sufficient.

2. Moreover, unsynchronized reads of the database can be permitted, as follows. By the special nature of the unification algorithm, each variable is bound ('written') at most once. That means:

a) Whenever a binding is read, *if* the value returned is other than Ω, that value is *necessarily* correct and final.

b) However, if the variable is seen to be unbound, any attempt to bind it must be done through a serializer which performs the required access and bind operation, but with serialization on a *per variable* basis.

c) In short, the liberalized access policy permitted by this special 'write once' property is 'read freely, queue to bind'.

3. Finally, the recirculating database version method can be criticized for excessive tuple copying as the stream of intermediate $\mathcal{I}_3$ representations is produced.

4.2 A More Liberal Solution

Suppose we seek to unify $\mathcal{T}_0 \circ \mathcal{I}_1 \circ \mathcal{I}_2$ with $\mathcal{T}_1$, the head clause of a target clause C_1. The 'write once' idea described above can be exploited as follows:

```
FUNCTION unify:[t1, t2, k2] =

    {s3 = makecells:['tuple, k2,
                allub],

    FUNCTION allub:i = unbound,

    FUNCTION termunify:[t1, t2] =
        IF   isvar:t1
        THEN trybind:[t1, t2]
        ELSE (* t1 is a function *)
        IF   isvar:t2
        THEN trybind:[t2, t1]
        ELSE (* two non variables *)
        IF   eq:[head:t1, head:t2]
        THEN argunify:[t1, t2, 2,
                tlength:t1]
        ELSE false,

    FUNCTION trybind:[var, newbind] =
        IF   eq:[tselect:[var, s3],
                        unbound]
        THEN (* bid to bind var *)
        {oldbind = ab:[var, newbind],
        RESULT
            IF   eq:[oldbind, []]
            THEN (* binding OK *)
                 true
            ELSE (* recur *)
                 termunify:[newbind,
                        oldbind]}
        ELSE (* already bound *)
             termunify:[tselect:
                          [var, s3],
                        newbind],

    RESULT [termunify:[t1, t2], s3]}
```

Fig. 4. Unification functions.

We create $\mathcal{F}_3$ as a tuple of *writeable cells*, equal in number to $|\mathfrak{N}(\mathcal{C}_1)|$ + UB(length($\mathcal{F}_1$), $\mathcal{F}_1$), i.e. the number of native variables in $\mathcal{C}_1$ plus the number of variables imported into this instantiation of $\mathcal{C}_1$. All entries in $\mathcal{F}_3$ are initialized to Ω.

The cells in $\mathcal{F}_3$ are read freely during unification, and, when seen to be equal to Ω, attempts to bind them are made as required.

A serializer procedure (pseudo functional) is created for each variable to service access and bind requests in the sense described above. Mutual exclusion within serializers is achieved by the *mutex* resource control construct described in (Jayaraman and Keller 1980).

Fig. 4 gives the basic functions involved in our unification approach. Note:

1. Literals are represented as nested tuples, with constants denoted as 0-ary functions. Hence the representation for the compiled clause in fig. 2 would be:

[p, V1, V2] :- [q, V1, [f, V3]], [r, V3, V2, [c]].

The top-level invocation is

`unify:[t1, t2, k2]`

where `t1` = $\mathcal{F}_0 \circ \mathcal{F}_1 \circ \mathcal{F}_2$, `t2` = $\mathcal{F}_1$, and k2 is the length of the desired $\mathcal{F}_3$ substitution tuple. The result is [*true*, $\mathcal{F}_3$] if the unification succeeds, and [*false*, *undef*] otherwise.

The internal function is `termunify` performs most of the required case analysis. If `t1` and `t2` are nonatomic, the auxiliary function `argunify:[t1, t2, a, b]` (not shown) attempts pairwise unification of {`tselect:[i, t1]`, `tselect:[i, t2]`}, for i ∈ {a, ..., b}. This is done in parallel, on a 'divide and conquer' basis, with eager failure reporting. We assume unique arities for each functor symbol.

2. `trybind` does unsynchronized reads of variable occurrences. If a variable is found to be already bound, *or* appears to be unbound but fails a binding attempt, `termunify` is called on the value retrieved and the rejected new binding.

```
s3m = make:['tuple, k2, allmut],

FUNCTION allmut:i = gmutex:[],

FUNCTION ab:[var, newbind] =
   (* access and bind serializer *)

   wait:[tselect:[var, s3m],
      IF   eq:[tselect:[var, s3],
                  unbound]
      THEN (* free to bind now *)
           seq:[treplace:[var, s3,
                  newbind],
                [1]]
      ELSE (* too late *)
           tselect:[var, s3]]
```

Fig. 5. Unification synchronizer (nested in `unify`).

$\mathcal{F}_0$ = p(X, a, U, Y)

$\mathcal{F}_0 \circ \mathcal{F}_1$ = p(X, a, f(Z), Y)

$\mathcal{F}_1$ = p(Z, X, Y, g(X))

$\mathcal{F}_1$ =

Ω	f(V4)	Ω	Ω
V1 (X)	V2 (U)	V3 (Y)	V4 (Z)

$\mathcal{F}_2$ =

V4	*undef*	V5	V6
V1 (X)	V2 (U)	V3 (Y)	V4 (Z)

$\mathcal{F}_3$ =

V4	a	f(V6)	Ω	g(V2)	Ω
V1 (Z)	V2 (X)	V3 (Y)	V4 (X')	V5 (Y')	V6 (Z')

Fig. 6. Sample unification execution.

4.3 Synchronization Control

The function **trybind** in fig. 4 relies on **ab:[var, newbind]** ('access and bind') to manage the writeable cells representing $\mathcal{F}_3$. Applications of **ab** return a null tuple [] if the requested binding was adopted; otherwise, the existing binding is returned. Fig. 5 provides the code for **ab**.

The following comments will be helpful in understanding fig. 5:

The tuple **s3** is parallelled by a tuple **s3m** of mutex data objects (each created by the primitive **gmutex:[]**). The FEL construct **wait:[m, exp]** ensures that at most one **exp** within a **wait** on a mutex **m** will be executed at a time. Hence a 'critical section' type effect is obtained.

The operation **treplace:[i, t, v]** is the write analog of **tselect:[i, t]**.

The pseudofunction **seq:[a, b]** causes the sequential evaluation of **a** and **b**, generally for their side effects, and then returns the value of **b**.

A sample application of **unify** is shown in fig. 6.

4.4 Parallelism Obtained

We claim, without rigorous proof, that the unification approach just outlined exploits as much concurrency as is possible within a straightforward manner. Observe in particular:

Argumentwise concurrency is attempted whenever two nonatomic terms are to be unified.

Since mutexes are implemented on Rediflow with individual server processes, no delays are experienced on **wait** operations unless two involve the same mutex (here, when two bindings of the same variable are attempted simultaneously). These delays seem inherent in the unification process.

4.5 Variable Exportation

We now return to the question of exportation of unbound variables in solution substitutions. The problem is comparable to that of variable importation during unification accomplished by $\mathcal{S}_2$, and a similar relocation technique suffices.

Suppose we are to compute $\mathcal{S}_1' = \mathcal{S}_1 \circ \mathcal{S}_2 \circ \mathcal{S}_3'$. Let n = UB(length($\mathcal{S}_3'$), $\mathcal{S}_3'$), the number of variables left unbound in $\mathcal{S}_3'$. If n = 0, we have no variable exportation problem. Otherwise, we define a vector s3rel, where s3rel[i] = V(length($\mathcal{S}_1$) + UB(i, $\mathcal{S}_3'$)), and extend $\mathcal{S}_1$ to include n new variables, all unbound. Then when references to unbound variables are detected in **bindterm**, they are relocated through s3rel.

4.6 Economic Issues

We now offer a brief economic analysis of this technique for representing binding environments. Two questions naturally arise when this method is considered for large scale logic programming applications:

1. Will variable importation cause substitution vectors to become unreasonably large, and

2. Will the repeated use of composition functions eventually degrade the speed of producing each subgoal solution?

We believe the answer to each question is *no*, but do not as yet have conclusive empirical evidence for support. However, we offer the following informal arguments.

Question 1: The size of each substitution vector is equal to the number of native variables in its associated clause, plus the number of unbound variables imported into its environment. If an imported variable does appear in a term bound to a native variable, that variable importation is necessary and useful under our technique for management of multiple environments. The wasteful case is when a variable is imported, but is in fact unreachable. Note such variables could be detected by complete traversal of the goal terms undergoing unification, but we judge this test to be unacceptably slow in practice.

Instead, we offer the following simple optimization. Each imported variable Vj in an instantiation substitution $\mathcal{S}_3$ will be the image of some Vi in the matching unification substitution $\mathcal{S}_2$. If Vj is unreachable, it will surely be still unbound when restatement of a solution $\mathcal{S}_3'$ takes place. Hence Vj can be mapped by s3rel back to Vi, rather than to a new Vj'. The net effect is that the number of potentially unreachable imported variables in a goal environment is proportional to the path length from that goal to the root query in the overall AND/OR tree (i.e. the number of 'parent goals'). Thus unreachable imported variables do not accumulate as we move left to right in the AND/OR tree.

Question 2: In examining the code of fig. 3, we see recursive traversal of terms in **apply2** and **restate**. While it is true that such traversals do cascade as we move to the right (and upward) in the AND/OR tree, we also point out that

a) such traversals are done only as genuinely required, given Rediflow's underlying lazy evaluation method, and

b) once such a traversal is done, its result is recorded in a substitution vector, thereby saving OR-siblings from the same effort.

5. CONCLUSIONS

We have presented a vectorized representation for substitution data objects as an efficient technique for environment representation when doing OR-parallel logic programming on applicative architectures. Procedures for maintaining these representations were outlined in the two situations of most interest: concurrent unification, and solution reporting.

This work has similar intent as do most storage management techniques within OR-parallel logic programming implementations. Of particular relevance is the work in (Ciepielewski and Haridi 1983a, Ciepielewski and Haridi 1983b). However, our work contrasts with theirs in the following respects:

1. Environment separation is accomplished incrementally as a preface to unification, rather than as bindings are performed.

2. All variables pertinent to a goal are collected in a single vector, which we believe will have locality advantages on distributed architectures.

3. No 'directory' or 'context' structures are used; vectorized substitutions suffice for all environment representations.

4. The method is integrated with a concurrent unification algorithm.

5. Finally, solution reporting (to AND-siblings or a parent goal) is done by a substitution composition technique which is both efficient and purely applicative, thereby facilitating additional concurrency in its execution.

ACKNOWLEDGEMENTS

The author is grateful for the insightful comments of S. Haridi and P. Roussel while preparing this paper. Also, the kind hospitality of Jack Dennis and the MIT Computations Structures Group during the preparation of this paper is gratefully acknowledged.

REFERENCES

Burge, W. H. Recursive Programming Techniques. Addison-Wesley, 1975.

Ciepielewski, A., and Haridi, S. A formal model for OR-parallel execution of logic programs. In R.E.A. Mason, editor, Information Processing '83. IFIP, Paris, 1983.

Ciepielewski, A. and Haridi, S. Storage models for OR-parallel execution of logic programs. Technical Report TRITA-CS-8301, Royal Inst. of Tech., 1983.

Clark, K. L., and Gregory, S. PARLOG: a parallel logic programming language. Technical Report DOC 83/5, Imperial College, May, 1983.

Conery, J. S. and Kibler, D. F. Parallel interpretation of logic programs. In Proc. Conf. on Func. Prog. Lang. and Comp. Arch., pages 163-170. ACM, New York, 1981.

Dwork, C., Kanellakis, P. C. and Mitchell, J. C. On the sequential nature of unification. Technical Report CS-83-26, Brown University, December, 1983.

Furukawa, K., Nitta, K. and Matsumoto, Y. Prolog interpreter based on concurrent programming. In Proc. First Int'l. Conf. on Logic Programming, pages 38-44. Marseille, September, 1982.

Haridi, S. and Ciepielewski, A. An OR-parallel token machine. Technical Report TRITA-CS-8303, Royal Inst. of Tech., May, 1983.

Jayaraman, B., and Keller, R. M. Resource control in a demand-driven data-flow model. In Proc. International Conference on Parallel Processing, pages 118-127. IEEE, August, 1980.

Keller, R. M. and Lindstrom, G. Applications of feedback in functional programming. In Proc. Conf. on Func. Lang. and Arch., pages 123-130. ACM, Portsmouth, N.H., October, 1981.

Keller, R. M. FEL Programmer's Guide. AMPS Technical Memorandum 7, Univ. of Utah, Dept. of Computer Science, April, 1982.

Keller, R. M. and Lindstrom, G. Toward function-based distributed database systems. Technical Report UUCS-82-100, Univ. of Utah, January, 1982.

Keller, R.M., Lin, F.C.H., and Tanaka, J. Rediflow multiprocessing. In Proc. Compcon '84, pages 410-417. IEEE, San Francisco, February, 1984.

Lindstrom, G., and Panangaden, P. Stream based execution of logic programs. In Proc. Int'l. Symp. on Log. Prog., pages 168-176. IEEE, Atlantic City, NJ, February, 1984.

Martelli, A. and Montanari, U. An efficient unification algorithm. ACM Trans. Prog. Lang. 4(2):258-282, April, 1982.

Shapiro, E. Y. A subset of concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, January, 1983.

Umeyama, S. and Tamura, K. Parallel Execution of Logic Programs. Technical Report, Electrotechnical Lab., 1982.

Warren, D.H.D. Implementing Prolog. Technical Report D.A.I. Research Report No. 39, Univ. of Edinburgh, May, 1977. Vol. 1.

Warren, D.S. Efficient Prolog memory management for flexible control strategies. In Proc. Int'l. Symp. on Log. Prog., pages 198-202. IEEE, Atlantic City, NJ, February, 1984.

A CLASS OF ARCHITECTURES FOR A PROLOG MACHINE

L. V. Kalé
David S. Warren
Computer Science Department
SUNY at Stony Brook
Stony Brook, NY 11794, USA

ABSTRACT

This paper presents a view of the computation of Prolog programs that is suitable for expressing parallelism. We develop an idealized architecture consistent with this view which allows for exploiting most types of parallelisms. The architecture is based on an efficient *broadcast* link. The idealised architecture requires infinite resources, and so we consider various ways of mapping it onto practical topologies. Types of parallelism that should be retained while making this approximation are discussed, and a class of architectures is developed that approximates the ideal. The parameters of this class are defined and criteria for evaluating them are given.

1. Introduction.

Prolog is becoming widely accepted as a powerful programming language. Its non-procedural formulation (van Emden 1976) and clean semantics (of pure Prolog at least) make it an executable specification language. A large number of AI applications were programmed in a relatively short time in Prolog (Szeredi 1982). The increased availability and low cost of hardware along with the increased demand for computational power makes it important to attempt to speed up Prolog using parallel hardware.

Prolog has certain properties that make it an attractive language for exploiting parallelism. The expression of parallelism is natural in Prolog. Multiple clauses for a single predicate allow for expressing OR-parallelism. The body of a clause consists of a conjunction of literals, and this allows for AND-parallelism. Although most Prolog implementations impose a left-to-right sequencing, for pure Prolog it can be considered as an optimization, implemented because in most cases (when variables are shared between literals), sequential execution is more efficient than independent execution of subgoals. Besides, in the absence of parallel hardware, there is little motivation for not imposing sequencing.

The criteria that we stipulate for an implementation of parallel Prolog are: (1) it should be realizable with current or foreseeable hardware. (2) it must be scalable, i.e., one should be able to add extra processing power to the system and get a gain in performance without a significant redesign of the system. Implementations based on shared global memory are not acceptable, because shared access to a common memory will take more time as the number of processors increases.

Several attempts have been made towards this goal. The AND-OR process model of (Conery and Kibler 1983, Conery 1983) concentrates on how to decompose a problem into its subproblems when there are dependencies across the subproblems. It is a process-based model of computation: questions of assigning processes to processors and the structure of communication links between the processors are postponed to a later stage.

The EPILOG system of (Wise

1982) deals mainly with the changes that need to be made to the language Prolog to make it a suitable candidate for implementation on a data-flow machine. Since Prolog as it stands now has constructs that are useful only in a sequential implementation, making changes in the language is certainly an important issue. We believe that the architectures for a Prolog machine could be investigated concurrently. An architecture should implement at least pure Prolog and should be flexible enough to incorporate extensions as needed.

The PRISM system (Kasif, Kohli and Minker, 1983) implements Prolog on a special architecture called ZMOB (Rieger, Bane and Trigg, 1980) They have a set of problem-solving machines and an additional set of processors to store clauses. Thus each unification requires two messages. The processors communicate via a single 'fast conveyer belt'. So the communication delays increase linearly with the number of processors, reducing the scalability.

This paper develops an approach that does not assume shared memory and deals with issues starting from the available hardware level through architectures, execution methods and control strategies. The next section presents a general view of Prolog computation that is suitable for parallel interpretation. In Section 3, we develop and optimize an idealized architecture for Prolog. We examine its properties and the way it can be used to execute Prolog programs in parallel. It is idealized in the sense that it presumes unlimited resources. In Sections 4 and 5, we exhibit ways of realizing this architecture with limited resources, identify the parameters of the class of architectures thus generated and examine the issues that are involved in selection of each of these parameters.

2. A View of the Computation.

Traditionally, a Prolog computation has been viewed as an AND-OR tree (Bruynooghe 1982) with the AND arcs corresponding to each literal of the query and the OR arcs corresponding to the possible clauses for each literal. Although elegant in some respects, this picture of computation hides its complexity in the requirement that all the substitutions must be *consistent* across the tree. Ability to view the subproblems independently is crucial to developing models that will execute them in parallel. Therefore we constrain our tree models so that each node represents a completely described subproblem that is solved without any reference to the nodes in the tree above it. We make the constraint more concrete by associating a *partial solution-set* (PSS) with each node in the tree. This set consists of substitutions for variables that make the subgoal represented by the node true, and is to be computed using only information from other nodes in the tree below it.

The tree, then, should represent the subproblem reduction process via the AND-arcs and exploration of alternative solutions via the OR-arcs. However, in a large number of cases, the AND-OR tree does not represent the subproblem reduction process faithfully. Consider an AND-node with the query: 'p(X),q(X,Y)'. In most practical implementations (parallel or sequential), this problem would be solved by solving one of the literals (say, p(X)) first, and solving the other literal *with the values for X provided by the first.* Thus, if $x_1, x_2 .. x_n$ are the values for X returned by p(X), the true subproblems of the problem, i.e. those that must be solved to solve the original problem, are: 'p(X)', 'q(x_1,Y)', 'q(x_2,Y)' .. 'q(x_n,Y)'. (assuming q(x_n,y) was the only q to succeed). This sub-division cannot be represented in the AND-OR tree: it has just one node for the literal q(X,Y). Given our constraint that a problem must be solved using only information from below it, the AND-OR tree dictates that the two problems, p(X) and q(X,Y) should be solved independently; the solution sets

would then be joined to get a consistent solution.

We therefore introduce a somewhat different picture of a Prolog computation. The computation is represented by a REDUCE-OR tree, similar to an AND-OR tree. The root corresponds to the query, and is a REDUCE node. Except the root, each REDUCE-node corresponds to one clause of the program. The sub-nodes of a REDUCE node are OR-nodes. They correspond to a set of subproblems that can help solve the problem that the REDUCE node represents. There may be multiple ways of reducing a problem to subproblems. However, the arcs correspond to one particular way chosen by the control strategy (CS). Thus, for example, if 'p(X),q(X,Y)' is the query, a possible structure for the root of the tree is as shown in Figure 1.a. A dot on top of a variable indicates that the literal containing the dotted occurrence is the *generator* of that variable. The values of that variable used in the subproblems for other literals are those that satisfy the generator literal. The generators are chosen by the CS. The CS might have dictated the structure shown in Figure 1.b, or another (Figure 1.c), where the parent node computes the join.

Each OR node corresponds to a single literal. The multiple arcs from it correspond to potential solutions to this literal. To make the picture more uniform, we will consider each clause of the program as a potential way of solving *any* literal. (As opposed to only those clauses that have the goal predicate as their head literal). All OR-nodes now have the same structure. Each has exactly N children, where N is the number of clauses of the program.

The computation can be viewed as a process of developing this tree. Starting with the main query as the sole REDUCE node in the tree with an empty PSS, one extends the tree in any of the following ways:

(1) Corresponding to any literal of an active REDUCE node R, one may add an arc from R to a new OR node O representing an instance of the literal, provided the generator literals for those variables that are not generated by this literal have already been attached to R. Then O is instantiated with a consistent composition of the substitutions, one from each of the PSS of the generator literals.

(2) To any OR-node that is a leaf of the tree, one may add N arcs to REDUCE nodes, one corresponding to each clause of the program. Each REDUCE node with a clause whose head unifies with the literal of its parent node is considered an *active* node. The root is defined as an active node. The instantiated body of the clause becomes the goal of the new REDUCE node (say R). If the body is empty (the clause is a 'fact'), the PSS associated with R becomes a singleton set with the unifying substitution as its only member.

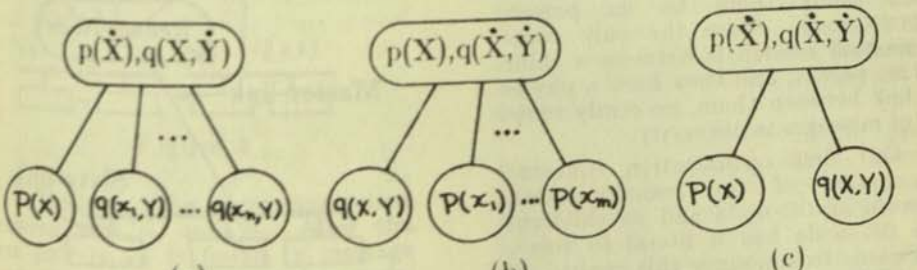


Figure 1

(3) Any entry from the PSS of a REDUCE node can be added to the PSS of its parent node. A substitution can be added to the PSS of a REDUCE node R (representing a composite goal G) if it is a consistent composition of the substitutions, one for each of the literals of G, from the PSS's of the OR nodes below R.

3. An Idealized Architecture.

We will develop an architecture for implementing an execution scheme based on the REDUCE-OR tree. Our first approximation is isomorphic to the tree itself, with a processor corresponding to each node and a physical communication link corresponding to each arc. In this section, we optimize this architecture step by step, and show how it can support various kinds of parallelisms.

First, let us describe the execution method and its properties on this architecture. The top level node gets the query and decides on the grouping and sequencing of the subproblems. It then sends the appropriate subproblems to the OR-nodes just below it, in a sequence consistent with the control strategy and the tree-development rules stipulated above. Note that we assume an arbitrarily large number of OR-nodes available to each REDUCE node. Each OR-node transmits the literal it received to all the REDUCE nodes that are its children. A REDUCE node sends any answer that it constructs, either by matching a fact or by combining solutions of subproblems sent to it, to its parent OR-node, which sends them to its parent REDUCE-node. Since the only communication needed is between a child and its parent, and they have a physical link between them, no costly routing of messages is necessary.

Our first optimization concerns the nature of the communication between an OR-node and its children. The OR node has a literal to solve, and needs to announce this goal to all its children nodes. Instead of sending an identical message on each of the links, it would be more economical if the message is *broadcast* to all the successor nodes at once. So let us replace the links from the OR node with a single link to an efficient broadcast channel to all the subnodes. For brevity, we will call such a channel a *net*. An *ethernet* (Metcalf and Boggs 1976) is an example of an efficient broadcast link. As this link is used to pose the problem to the net and to collect the answers back from it, we will call it the *master* link to the net. The interconnection structure around an OR node now looks as shown in Figure 2.

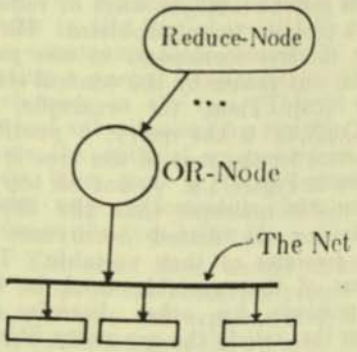


Figure 2

Let us now examine the work that an OR-node needs to do. It gets (a message corresponding to) a goal literal from its parent node and broadcasts that onto its net. It then watches for any solutions appearing on the net and sends them back to the parent. It really acts as a front-end to

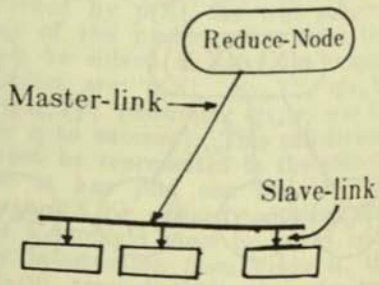


Figure 3

the REDUCE node. We therefore eliminate the OR-nodes altogether. The modified structure is shown in Figure 3.

Now we have only one type of nodes in the tree. Each node has a single clause. It receives a literal to be solved from the net and tries to use its clause to solve it. If the head of its clause unifies with the literal, it becomes the manager of the (possibly empty) sub-query consisting of the right hand side of the instantiated clause. It then invokes the control strategy to decide the grouping and sequencing of the subproblems. Using this, it communicates the subproblems in appropriate order to different nets via its master-links. For each solution to the subproblem obtained from a subnet, it either starts new subproblems that were waiting for the value of a variable provided by this solution, or combines the answers of subproblems to form a solution to the original problem. It sends each solution so obtained to its parent node.

Our next optimization is really a generalization to allow more flexibility for the control strategy. It concerns communication needs across the subproblems. Consider the query: 'p(X,Y),q(X).'. Let p be the generator of X and q a filter of X. In our current version, the two problems will be solved on different nets as shown in Figure 4.

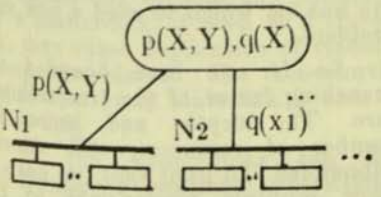


Figure 4

Consider the communication between nets N_1 and N_2. After the first pair (x_1,y_1) is found, a message goes from N_1 to the parent node, which sends another message to N_2. These two messages could be avoided if N_1 and N_2 were the same net and were given the joint problem: 'p(X,Y),q(X)'. An algorithm for executing such problems on a single net is described in (Warren et al. 1984) . Note, however, that this groups together the functions of all the N_i's, thus potentially reducing the parallelism between solutions of the $q(x_i)$'s: all the $q(x_i)$'s *have* to be solved on the same net now. As it is not always beneficial to solve the composite query on a single net, it should be the prerogative of the control strategy to pose either a composite multi-literal problem or a single literal one to a net.

3.1. Opportunities for exploiting parallelism.

We now examine how different kinds of parallelisms can be exploited on this architecture. The discussion here will help our search for practical implementations of the architecture in the next section: they should try to retain as much of this parallelism as possible.

The *AND parallelism* involves evaluating two or more literals of a composite goal simultaneously. As a node has master links to an arbitrarily large number of nets, this parallelism can be easily implemented on our architecture. When the literals share variables, it is not always efficient to compute them in parallel. So it is left to the control strategy (CS) to choose whether to execute the literals in parallel or not.

The *OR parallelism at the literal level* involves exploring all the solutions to a given literal simultaneously. As the literal is broadcast on a net and all the nodes start working on it at once, this parallelism is inherent in our architecture.

The *OR parallelism at the query level* is the OR-parallelism across literals which is exhibited by non-deterministic predicates. Given two literals with common variables, it involves starting execution, in parallel, of each *instance* of the second literal for every solution of the first. For

example, if the query is 'p(X,Y),q(Y,Z)' and p has multiple solutions, then we *can* start exploring each $q(y_i,Z)$ as soon as p returns a new Y value (y_i). Although in our idealized architecture it is always beneficial to do so, in practical models, where the nets and the nodes need to be shared, it may not be so. In such models, the usefulness depends on features of the problem (program and query) and the specific topology of the architecture, which are best handled by the CS. As the solutions are always sent to the parent node, (which can then decide to start a new process for the next literal), this parallelism is implementable on this architecture, although the decisions to do so is left to the CS.

The *LOOKUP parallelism* involves looking up the clauses in the database in parallel. There is a distinction between this and the OR parallelism at the literal level. There, one just needs to *start* parallel processes, corresponding to different matching clauses, after having looked up those clauses in the database (program). In our model, as the database is distributed across the nodes of a net and all of them receive the literal to be solved simultaneously (via a broadcast), the lookup is automatically done in parallel.

4. Approximations and Practical Topologies.

The basic problem with the idealized architecture is that it assumes infinite resources. In particular, it assumes:

(1) an arbitrarily large number of nodes on each net (one for each clause);

(2) an arbitrarily large number of master-links from every node to the nets, one for each subproblem that the node needs solved; and

(3) an arbitrarily large number of nets (as the computation tree could be arbitrarily large).

Clearly, in practical topologies, the nodes, nets and the links have to be shared.

Firstly, we have to limit the number of nodes on a net to a fixed number. Thus, we must allow a single node to have more than one clause. Some of the advantages of parallel lookup are reduced by doing this, but with a careful distribution of clauses it can (in most cases) be brought at the same level as before. It has a further advantage of effective resource utilization: the idle time of a node is reduced because a set of nodes will remain unused in the original model if the predicate that they represent does not figure in the computation. With multiple clauses at each node, the chances are better that some clause on any given node will be used in the computation.

Secondly, we limit the number of master-links coming out of a node to some fixed number. Now, more than one subproblem may have to be sent via the same master-link onto the same net. The nets may have to solve more than one goal concurrently. The answers coming back on the net from individual nodes have to be labeled so that the master can recognize them as answers to a particular query. The control strategy (CS) at each node must take into account the fact that the nets it subcontracts may already be working on some other problems. Thus it must keep track of the tasks that it has assigned to each of the sub-nets, and load-factors of the nets. It could use this information to choose the link on which to send a new subproblem.

So far we have bounded the branching factor of the tree architecture. The depth, and hence the number of resources, still remains unbounded. To limit that, we have to allow multiple master-links to the nets. With such links a net is responsible not just for multiple subproblems from the same master-node, but multiple subproblems from multiple master nodes. This allows for cyclic structures in the topology, and thus sharing of nets for different goals. It also brings our architecture into the domain of

the physically implementable. This is the step in which we map the (infinite) tree architecture onto a finite interconnection network. As we shall see later, this mapping can be done in a variety of ways, leading to a variety of architectures.

Before proceeding to discuss some of the specific architectures with the above properties, let us examine the essential features of this class of models. The network consists of a number of nets and processors. There are two kinds of connections between a processor and a net: master-link, through which a node poses a problem to a net, and slave-link, through which a node gets a literal to be solved. The set of clauses on all the slaves of a net are exactly the clauses of the program, without any duplication. i.e. a net is a *complete* problem-solver.

We can make further simplifying assumptions and optimizations which will help us categorize the possible networks. Firstly, we assume that each node has the same number of slave-links. Secondly, the same physical link may be used as both a master-link and a slave-link to a node. However, there is an important asymmetry here. A node may have a master-link to any net whereas it may have slave-links to only those nets that do not have slave-links to another node with the same clauses as it has. Thus, a slave-link can be used as a master-link without any problem, but a master-link, if used as a slave-link, may cause duplication of clauses among the slaves. We henceforth assume that every slave-link is also a master-link.

As the master-links can be added/removed without restriction, whereas the slave-links are subject to the above restriction, it seems reasonable to categorize the possible topologies according to the slave-links first. A *diagram* of a topology depicts all the nodes, each labeled with a number denoting its set of clauses, all the nets and all the master and slave-links between them. A *skeleton* is a diagram without the master-links. The skeleton shows how the problem solvers (the nets) of a topology share the resources (the processors). The diagram shows how they can communicate with each other.

4.1. Topology of the Slave-links: the Skeleton.

We now examine the different kinds of structures of skeletons possible. Each of them gives rise to a series of topologies. The skeleton-structure decides the scalability and strongly affects the performance. By scalability we mean the ease with which the network can be expanded. Following (Reed 1983), we will measure the scalability in terms of the minimum increment of processers needed to move to the next bigger topology in the series. For the purposes of this section, we will ignore the effect of control strategy (CS) on performance, i.e. we will compare the performance of different skeletons assuming optimal CS on each one. Given a fixed number of nets, the performance of an architecture which has a completely-connected net-graph can be considered optimal, because it can use its resources effectively: no net need remain idle while others are overloaded. We will use this ability to spread work evenly among the nets as the criterion for comparing the performance of two architectures. Architectures on which the problems do not have to wait for resources when resources are available *somewhere* on the network are superior to those on which topological reasons force the problems to wait. More accurate comparisons can be made by simulation studies. (The performance is also affected by the master-link structure. However, as we are designing the skeleton before the master-link structure, it is important to assess the effect of the skeleton-structure on performance).

The reason that there is a multiplicity of possible skeletons is that a node may be a slave of more than one net. The simplest skeleton is the one which allows exactly one slave-link from each node. This leads to a collection of isolated nets. The nets communicate via the additional master-links that connect the nodes of one net to another net. This skeleton has some important properties that the others do not possess:

1. The distribution of clauses on each net may be different.
2. Even the number of nodes on each net may be different.
3. It is scalable without any restrictions, as there are no dependencies across the nets.

One need add one net and/or one processor to extend it.

When we allow 2 or more slave-links from a node, the situation is much more complex. The reason that we want to consider this option is that it allows increased processor sharing and thus tends to avoid processor idling.

It is still *possible* to have different distributions of clauses on different nets. However, it is extremely difficult to design a consistent system with such distributions. Also, there are no obvious advantages to doing so. Particular situations in which different distributions are optimal may exist, but it would be very difficult to take advantage of these in a general purpose system. Therefore, we assume that all the nets have identical distribution of clauses.

In the following discussion, let c be the number of slave connections per node, n the number of nets, p the total number of processors and k the number of clause-groups (i.e. the number of slave processors) on each net. Assume that k is fixed by other considerations (discussed in the next section). Notice that there are k different *types* of processors in the system, in the sense that they have k different sets of clauses. We will label each processor with a number $i, 0<i\leq k$, corresponding to its type. We will present a few series of topologies, and compare their performance and scalability.

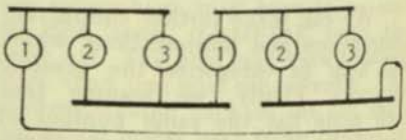


Figure 5

A topology of the first series is depicted in Figure 5. Any specific topology of this series can be easily extended by adding 2 nets and k processors to it. The second series is a grid-like structure. Given $n=2*m$ nets, one constructs this topology by laying

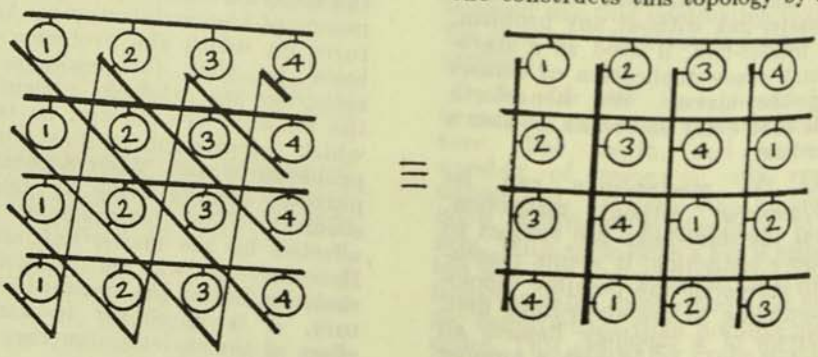


Figure 6 $n = 8, k = 4$

out the m nets as in an isolated topology, thus laying out all the processors. The nets are numbered 0...$m-1$. One then adds the remaining m nets one by one, connecting the j'th new net to the i'th node from the $(i+j \bmod m)$'th old net. A few example nets of this series are shown in Figures 6,7 and 8. Again, the minimal increment is 2 nets and k processors. However, this series has a better performance than series-1. The largest distance between two nets (the number of nodes that have to be visited in order to go from one to the other) in the first series is approximately $n/2$. In the second series, for $n \leq 2*k$, the distance is 2. In general, it is approximately $n/(2*(k-1))$. It thus seems that activation would tend to spread more evenly among the processors in a series-2 topology. To see this point more clearly, consider the graph of a topology with the nets as nodes and a path from a net through a node to another net as an arc between them. This arc represents shortest communication path between two nets. Figure 9 shows the graphs for a topology of each series. For the graph of a series-2 topology, the minimal spanning tree is much bushier than that for a series-1 topology (with the same number of nets). Thus, within a given (small) number of steps, one can reach more nets in series-2 than in series-1.

A series-2 topology (a grid) with $n=2*k$ yields a regular grid. An example is shown in Figure 6. Notice the

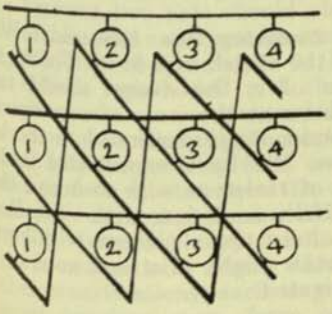
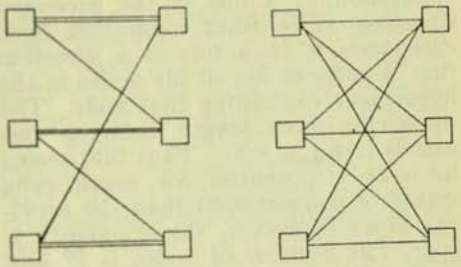


Figure 7



Series-1 Series-2

Figure 9

labeling of node-types in the net. One could obtain the labeling by labeling all the nodes on one net arbitrarily (say in the order 1 to k), and then labeling the nodes in the adjacent parallel net in the same order, but shifted one position in one direction. This labeling strategy is easily generalized to higher dimensions corresponding to higher values of c. Thus spanning-bus hypercube (SBH) architectures (Wittie 1981) are also included in our class of architectures.

The dual-bus hypercube architecture (DBH) (Wittie 1981) provides another interesting series of topologies. The processors are arranged as lattice points of a D-dimensional hypercube (with width k). The labeling is same as in SBH. But it requires only two connections per processor. Thus it is cheaper than the SBH. There is a preferred dimension. Each node has a

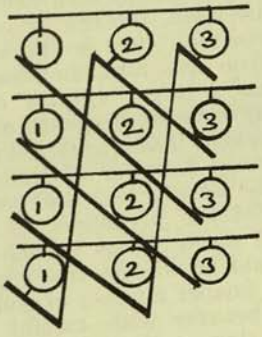


Figure 8

connection to a bus in the preferred direction. The other connection from each node is to a bus in a direction that is uniform for all the nodes in the hyperplane containing that node. The worst-case path length between two nets is $(2*\log_k n - 1)$. Thus DBH seems to have a potential for much even spread of computation than the previous series. However, the scalability is poor. The number of nodes is k^D and so the minimum increment is at least $k^D(k-1)$. Thus we see here a genuine tradeoff between scalability and performance.

5. Design Issues.

The brief analysis in the previous section prepares us for tackling the problem of designing a parallel execution system for Prolog. We now consider the design choices involved. Notice that we are not presenting solutions to the design problems here. Rather, we are attempting to list the choices and the issues involved in making those choices, so as to set a framework for future research.

5.1. Choosing k, the Number of Nodes per Net. The bandwidth of the net puts an upper bound on how many nodes can be put on a net. The tradeoff, within that limit, involves two factors. If the number of nodes on a net is increased continuously, a point may be reached when a large number of processors on the net tend to remain idle. Then, it would be more cost effective to use fewer nodes and use the extra nodes with new nets. On the other hand, too few processors per net mean reduction in the amount of parallelism possible, because now a node has a larger number of clauses and may have to deal with more goals concurrently. The decision depends on the average number of goals the net would be solving at a time and on the nature of the predicates of the program. If the topology is such that the net would be solving fewer goals at a time, a smaller number of nodes would suffice because with careful distribution of clauses, there is less chance of overloading a node. Also, if the system is meant for data-base applications, there would tend to be a large number of clauses for a single predicate. Then, a large number of nodes helps retain the parallel look-up. In a system for executing typically deterministic *programs*, there would be fewer clauses per predicate and a very few (typically one) of them would succeed beyond the initial guard literals of the clause. Then a small number of nodes (e.g. 3 or 4) would suffice unless more are needed because the net tends to be engaged in solving a large number of goals.

5.2. Choosing c, the Number of Slave-links per Node. A processor has to analyze every message that is broadcast on a net of which it is a slave as opposed to only those addressed to it from a net of which it is a master. Depending on the bandwidth of the net and the cycle time of the processor, one would get an upper bound on c, assuming continuous broadcasts on all the nets. Loading of the processor and cost of the links are the other factors limiting the value of c from above. The fundamental choice, though, is between $c=1$ and $c>1$ (mainly 2 and 3). With $c=1$, we get easy scalability and the ability to redistribute clauses on individual nets. We can then consider each net as an abstract independent problem solver. The cost, of course, is lesser utilization of processors. This is a *qualitative* choice, and at this point, it is unclear which one would be 'better'.

5.3. Designing the Skeleton. With $c=1$, the skeleton is fixed. For higher values of c, the design should take into account the issue of connectivity of the topology mentioned in the last section. We have enumerated a few series of skeletons with $c=2$ and they have their analogues with $c=3$. However, alternative structures with better properties might exist and need to be investigated.

5.4. Selecting the Number of Master-links per Node. We have already said that each slave link should be used as a master link. The question, then, is should there be additional master links. Distributed execution of recursive predicates is not possible without additional master links. A node that has the recursive clause broadcasts the recursive subproblem to one of the nets of which it is a master. If it is also a slave of that net, it will be the only one with that clause, (because duplication of clauses is not permitted) and will have to solve the subproblem itself. Therefore we expect that additional master links will be beneficial. The number of additional links is again limited by the cost per link and the cycle time of the processor. A point to note is that the master link is much more lightly used than a slave link; only the unicast messages carrying the answers need be considered by the processor. Also, with higher number of master-links one gets more even distribution of activity across the network.

5.5. Designing the Interconnection Structure of the Master-links. The design should provide for fast and even spread of computation across the network. In particular, a single net should not be overloaded (in comparison to others) and thus cause a bottleneck in the computation. This entails that all the nets should have about the same number of masters. The number of paths (of length 1, 2 etc.) between two nets should also be comparable. As an example, consider the skeleton shown in Figure 6. If the master-links were added such that all the processors on a horizontal net have one master-link to the next horizontal net, the activity from one net would tend to cluster onto the other. Randomly connected master-links (or carefully designed) may have more uniform connections among the nets. (Other considerations, such as easy to connect topology may, force one to implement the first structure)

5.6. The Control Strategy. Once a node receives a goal message from one of its slave links, it must attempt to unify it with the head of each relevant clause. For each successful unification, the control strategy (CS) has to manage a new query consisting of the body of the instantiated clause. It must consider the load on each of the nets of which it is a master and the control information associated with each clause that it manages. This control information may be provided by the user and/or obtained at compile time. It includes such factors as whether the predicates involved in the clause are deterministic and the functional dependencies among the variables of the predicates etc.

Using this, it must decide (a) how to subdivide the query corresponding to the body of the clause into subproblems and (b) which net to use for each subproblem. The former involves deciding whether to divide it in literals or in larger chunks and also deciding what sequencing of the subproblems is to be implemented.

The object of the CS is to optimize the performance of the whole network. As that may depend on the topology of the network and the CS has access to only the local information, we are faced with two options. Either we could have the topological information built into the CS or we could make it independent of the topology (and hope that it works well on most topologies). An option in another dimension is either to have the identical complete CS reside on each node or to let the control information be compiled into the representation of the clause itself (leaving only a simple executor at each node). The latter course seems faster and requires less storage on each node *if* there are few clauses on each node. It does require compile time analysis.

Another important task of the CS in any practical implementation would be to deal with the priorities of the subtasks that it manages. It could dictate the priority of a new subquery when it is broadcast and change it as the computation progresses. For example, it may reduce the priority of a subproblem after it has returned an answer.

6. Conclusion

We have considered only pure-Prolog in this paper. Ways of implementing the impure features of Prolog that are both required and useful need investigation. In particular, failure detection schemes are necessary for implementing *not* and *setof*. The semantics of side-effects (as in *write* etc.) under OR-parallelism has to be developed. Ways of updating the program have to be implemented. Unique resources (such as a printer), and duplication of clauses on a net need to be handled.

In conclusion, the basic bus-architecture approach seems promising. We envisage huge search-type problems being solved faster on a machine based on the architectures proposed here. The design choices described in the previous section are difficult ones. The criterion in making them is optimality *over a range of Prolog programs*, as opposed to optimality over a specific program. Thus, analysis alone will not be able to dictate the choices. We propose to do simulation studies of the various models generated by different choices.

REFERENCES

Bruynooghe, M. "The memory management of Prolog implementations", in *Logic Programming*, K.L. Clark and S.-A.Tarnlund, (eds.), Academic Press, New York, NY, 1982, 83-98.

Conery, J.S. and Kibler, D.F., "AND Parallelism in Logic Programs", *Proc.8th IJCAI*, 1, (Aug 1983), .

Conery, J.S., "The And/Or Process Model for parallel Interpretation of Logic Programs", Ph.D.Thesis, University of California, Irvine, California, June 1983.

Kasif, S., Kohli, M. and Minker, J., "PRISM: A parallel inference system for problem solving", *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, August 1983, 544-546.

Metcalf, R. and Boggs, D., "Ethernet: Distributed packet switching for local computer networks", *Comm.ACM*, 19, (July 1976), .

Reed, D.A., "Performance Based Design and Analysis of Multimicrocomputer Networks", Ph.D.Thesis, Purdue Univ., May 1983.

Rieger, C., Bane, J. and Trigg, R., "ZMOB : A Highly Parallel Multiprocessor", Tech.Rep.-911, Dept.of Computer Sc., University of Maryland,, College Park, MD., May 1980.

Szeredi, E.S.P., "Prolog Applications in Hungary", in *Logic Programming*, 1982, 19-31.

van Emden, M.H., "Programming With Resolution Logic", in *Machine Intelligence*, vol.8, 1977, 266-298.

Warren, D.S., Ahamad, M., Debray, S.K. and Kale, L.V., "Executing distributed Prolog programs on a broadcast network", *Proceedings of the 1984 Logic Programming Symposium, Atlantic City.*

Wise, M.J., "A parallel Prolog: the construction of a data driven model", *Proceedings of the 1982 Conference on Lisp and Functional Programming*, 1982, 56-66.

Wittie, L., "Communication Structures for Large Networks of Microcomputers", *IEEE Transactions on Computers*, C-30, 4 (April 1981), 264-273.

AN ARCHITECTURE FOR PARALLEL LOGIC LANGUAGES

J.A. Crammond and C.D.F. Miller
Department of Computer Science
Heriot-Watt University
79 Grassmarket
Scotland

ABSTRACT

The outline of an architecture to support the parallel execution of logic languages is presented. The implementation of a particular language, Parlog, is considered; attention is given to its "don't care" non-determinism which allows both and- and or-parallelism and returns only one solution.

The main features described are the control structure and the binding environment. The proposed control structure uses processes that build an and/or tree tailored for guarded clauses. For the binding environment we introduce a unification algorithm which solves the problems of multiple occurences of an instance of a variable in guards.

1. Introduction

A growing number of languages are being developed for specifying the parallel execution of logic programs. This paper outlines an architecture to support such languages.

Most parallel logic languages are based on sequential Prolog, and have the same or very similar declarative reading but different procedural semantics. The left to right evaluation of subgoals within a Prolog clause may be replaced by solving them in parallel; this is known as *and-parallelism*. The sequential order in which alternative clauses are tried in Prolog may be replaced or augmented by the ability to try all alternatives in parallel; this is *or-parallelism*.

It is possible to execute logic programs using or-parallelism and limited and-parallelism without additional language control facilities (Haridi and Ciepielewski 1983, Furukawa et al. 1982, and Conery and Kibler 1981) However, control facilities to specify some ordering of clauses can improve efficiency of or-parallelism by pruning the search tree (Kasif et al. 1983). The main problems arise with and-parallelism when two or more goals contain terms which share an uninstantiated variable, since only one of these goals should be allowed to instantiate it. The languages that allow limited parallelism usually force goals which share variables to be executed strictly sequentially, but allow goals with no shared variables to be executed in parallel.

Parlog (Clark and Gregory 1984) is a successor to their earlier relational language (Clark and Gregory 1981). (Parlog has itself undergone major changes since first described (Clark and Gregory 1983)). It solves the problems created by and-parallelism by using "mode declarations" to define which goal is the *producer*

of a variable's value and which goals are its consumers. Parlog allows both and-parallelism and or-parallelism to solve relations, in which only one solution is returned; it uses sequential-and with either or-parallelism in "eager" mode or sequential-or (i.e. like Prolog) in "lazy" mode to solve set-expressions, in which some or all alternative solutions are found. A comprehensive description of the language can be found in Clark and Gregory (1984).

In this paper we shall concentrate on relations, leaving consideration of set-expressions to a subsequent paper.

Our aim is to design a multiprocessor architecture able to support efficient implementation of all of the features of Parlog and also sufficiently flexible to be able to support other languages such as Concurrent Prolog (Shapiro 1983) and, perhaps, "normal" sequential Prolog.

Two of the main design features which we describe are the control structure and the binding environment that is used.

2. Basic Underlying Machine

The basic components of the abstract machine are a finite set of processors each with access to a shared global memory and also (optionally) to some local memory.

Global memory is divided into three sections: static memory contains the compiled code of the program; dynamic memory contains the various environment bindings produced during program execution; process memory contains information for each process created during execution.

A program is executed by creating processes to execute goals. These are allocated to a finite number of processors by a scheduler, running on a dedicated processor.

The architecture is control driven (Treleaven et al. 1982). Parlog offers flexibility in the ways in which goals can be executed (e.g. mixing sequential and parallel calls and clauses) and this is easily catered for with a control architecture.

3. Control Structure

The control structure is a hierarchy of processes representing the and/or tree which represents the search tree for satisfying a goal.

There are two types of node in this tree corresponding to two types of process: and-processes and or-processes.

An and-processes terminates with failure if any of its child processes fails. All of its children must succeed for it to do so. An or-process terminates with success if any of its children succeeds. Thus all of its children must fail for it to do so. (A child process of some process is one which has the given process as parent. The child is frequently created by the parent, but may be 'adopted', as described below).

The execution of a Parlog program begins with an and-process which executes the top level query. A child process is created for each goal specified in the query. These child goal calls are or-processes.

There may be a number of clauses composing the relation for each goal. A goal call will try each alternative clause by creating an and-process for each one.

Each of these and-processes will first of all attempt to unify the arguments in the goal call

with the arguments in the head of the clause. There are three possible outcomes:

1. Unification fails, causing the process to terminate with failure.
2. Unification suspends (an attempt was made to bind an uninstantiated input variable to a non-variable term); the process becomes input-suspended.
3. Unification succeeds; the process continues execution by trying to satisfy the guard clauses.

When an and-process has created child goal calls for its guard, it will suspend until they have terminated with success. If any of these children fails so will this process (since it is an and-process).

When reactivated, it will attempt to commit the goal call to this clause. This can have two* outcomes:

1. Commit fails: some other candidate clause committed first. Hence this process terminates with failure.
2. Commit succeeds: the process then continues, executing the clause body.

In case 2, the goal call is reduced to the execution of the body goals. This is reflected in the process tree structure: the and-process creates or-processes for the goals in the body which have the same parent as the calling goal or-process (see figure 1).

Once these body calls are created the and-process terminates with success, and hence its parent or-process also terminates.

* "Bounded buffers" that can cause commit to suspend (Clark and Gregory, 1983) are no longer in Parlog.

```
A1 :- G1, G2 | B1, B2 .   (1)
A1 :- G3, G4 | B3, B4     (2)

T : A1, A2 .
```

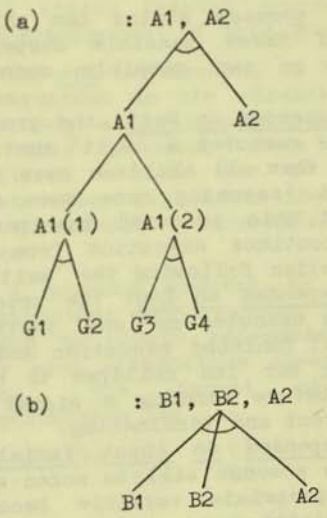


Figure 1: State of control tree (a) before a clause commits and (b) after the clause terminates.

3.1. Process Information

Each process created must carry sufficient information to execute some part of the program and to communicate its outcome to its parent. This information makes up the process control block. Included in this block are the following fields:

Process type: (AND or OR).
Code pointer: pointer to currently executed instruction.
Process pointer: pointer to this process.
Parent pointer.
Child pointer: most recently created child process.
Sibling pointer: sibling process previously created.
Process status.

A null child/sibling pointer indicates that this process currently has no active children/siblings.

The process status can signal one of three possible suspended states or two possible runnable states:

1. Suspended on Wait: the process has executed a 'wait' instruction. When all children have terminated (assuming none have terminated this process) the parent will continue execution from the instruction following the 'wait'.
2. Suspended on End: the process has executed an 'end' instruction; it finished execution and is waiting for its children to complete before sending a signal to its parent and terminating.
3. Suspended on Input Variable; the process will be woken when the appropriate variable becomes instantiated and will continue executing the same (unification) instruction that caused it to suspend.
4. Runnable, Queued; the process is runnable but has not yet been allocated a processor. It is held in a queue of runnable processes.
5. Runnable, Executing; the process is actually executing on a processor.

3.2. Control primitives

Here we describe primitive actions for handling processes. These actions require write access to the process memory and must therefore have some locking mechanism to avoid indeterminate results.

Create creates a process block containing a new process. The fields described above are initialised. Once created the process becomes "Runnable, Queued".

Fail terminates this process, killing any remaining child processes that may still exist. Either of the following conditions will result in the parent being 'failed':

1. The parent process is an and-process.
2. The parent is an or-process in "Suspended on End" state and this process is the last remaining child.

If the parent is an or-process in "Suspended on Wait" state and this is the last remaining child then the parent process is woken up.

Succeed terminates this process, killing any remaining child processes that may still exist. One of the following conditions will result in the parent being 'succeeded':

1. The parent process is an or-process.
2. The parent is an and-process is "Suspended on End" state and this process is the last remaining child.

If the parent is an and-process in "Suspended on Wait" state and this is the last remaining child then the parent process is woken up.

Kill children kills all descendants of this process.

Notice that the action of the child process upon success or failure depends not on its process type but on its parent's. Thus and-processes can be children of and-processes, a feature which can be useful for optimisation, as described below. The type of a process is determined by the instruction that was used to create it.

The above primitives are essentially built in to the control instructions described in the following section.

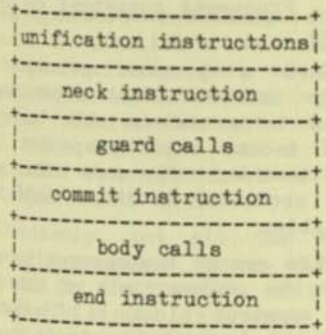
3.3. Control instructions

The instruction set developed for the abstract machine is based on Warren's PLM instructions for Prolog (Warren 1977).

A clause of the form

H :- G | B

will be translated to code of the form:



Guard and body calls are executed using the same instruction:

```
call    caddr, paddr
<arguments>
paddr:
```

This is interpreted as "create a new or-process which will begin execution at address caddr The 'old' process will continue at paddr. Following the call (between the call and paddr) are the call arguments.

The commit instruction separates the guard calls from the body calls in the clause. (If no guard is specified then it is executed after the neck instruction). It is interpreted as "wait for the guard's calls to complete (successfully) (i.e. go into "Suspended on Wait" state) and then attempt to commit the parent goal to this clause; if this succeeds then continue to the next instruction, otherwise the process is to terminate".

The commit instruction will also set an 'ancestor' field in the process control block so that the subsequent call instructions will attach created processes to the grandparent of this process.

The end instruction is the last instruction in the clause. Since the body goals do not get linked to this process, there are no children to wait for so the end instruction will cause the process to terminate (successfully) at once.

A process created by a call instruction will execute instructions to create and-processes to try each alternative clause for a goal. The format of these instructions is:

```
start:  try     C1
        try     C2
        ...     ..
        try     Cn
        end
```

The try instruction at "start:" is interpreted as "create a new and-process that will start execution at C1 (the start of clause instructions for the first clause)". The new process is always the child of the 'old' process in this case. The 'old' (parent) process continues at the next instruction after the try.

The end instruction will put the process into "Suspended on End" state and wait for its children. If there are no children to wait for it will cause immediate termination (failure).

The call and try instructions are suitable for executing goals and clauses in parallel. However, Parlog allows goals to be executed sequentially and clauses to be tried sequentially. This is achieved by the wait instruction. The sequence:

```
call    p
```

```
        wait
        call    q
```

will call goal p, then wait for it (and all its descendants) to terminate (with success) then call goal q. (This still allows subgoals of p to be executed in parallel). The conjunction "g1 , g2 , g3 & g4 , g5" will compile to:

```
call    g1
call    g2
call    g3
wait
call    g4
call    g5
end
```

This will execute g1, g2 and g3 in parallel and wait for them all to complete and then execute g4 and g5 in parallel.

The wait instruction can be used similarly when selecting alternative clauses:

```
C1 ; C2  =>  try    C1
             wait
             try    C2
```

The five instructions call, try, commit, wait and end are the basic control instructions. For convenience and efficiency they can be combined to give the following instructions:

```
sequential call:
   call l1,l2 + wait => scall l1,l2
sequential try:
   try C1     + wait => stry C1
last goal call:
   call l1,l2 + end  => lastcall l1
last clause to try:
   try C1     + end  => lasttry C1
```

Two other instructions are introduced for the special cases when only one clause exists for a goal and when only one body goal exists for a clause.

```
onlycall C1
```

will, instead of creating an or-process, change itself into an or-process and execute the code at C1.

```
onlytry C1
```

will, instead of creating an and-process, change itself into an and-process and execute the clause at C1.

3.4. Processes suspended on variables

When a process attempts to unify uninstantiated input variables to a non-variable term it will become input suspended, and must then wait for some other goal (the producer) to instantiate that variable.

This can be implemented by setting the status field of the process control block to "Suspended on Input Variable", and by having a channel field in the block which contains the address of the (dereferenced) variable on which the clause suspended.

When a process unifies that variable with a term any process sleeping on this variable will be reactivated. This involves checking for suspended processes when a clause commits (and thus makes its instantiations public).

To minimise the overhead of this checking the channel field could be stored separately, in appropriately indexed tables (or hash tables); alternatively an associative memory could be used.

4. Environments

An environment consists of frames which contain the bindings of the variables of a clause for its current call.

In sequential Prolog, only one environment is accessible at any given time during execution, because of the sequential

execution. Thus only one occurrence of an instance of a variable can exist at one time.

When or-parallelism is introduced more than one occurrence of an instance of a variable may exist at the same time - if a goal is called which invokes three clauses to be tried, then there will be three different instances of the goal argument variables.

In general, for each clause invoked in parallel a new environment is required. Each of these environments will be an (exact) copy of the environment of the goal call together with a local frame for the value cells of the variables in the clause.

Obviously, not all the variables in ancestor frames will be affected by the results of unification of goal arguments and clause head arguments. It could be possible for those frames unaffected by unification to be shared. Once the call commits to a clause the calling clause will inherit the new environment to replace its old one.

And-parallelism poses another problem in that a number of goals may access the same call frame at one time. In particular, different goals may update different variables in the same frame so that other (parallel) goals can read the resulting binding. Therefore, rather than replacing an entire frame when a goal commits, only the values of affected variables are copied back into the shared call frame.

Hence, a clause must hold private copies of variables which it alters in unification. As a further complication, guard goals and their descendants must access these private copies of variables rather than the public ones.

With the conventional unification algorithm used for Prolog (Warren 1977, Bruynooghe 1981) and unification of two uninstantiated variables results in the more recent (i.e. the clause variable) being assigned a reference to the older one. Subsequently, a variable in the call arguments of a goal may dereference to a variable in any ancestor goal in the environment.

This is unsuitable for a parallel system. Thus we have developed a unificiation algorithm that constrains call arguments to dereference only to variables in either the call frame or the local frame, with the exception of input (read only) variables. This restriction even applies to complex terms.

4.1. Unification

Unification has three stages:

1. Unification of call arguments with clause head arguments. The values of the variables used in the call are copied from the call frame to a (local) frame called the output frame. These may contain undef, terms, or references to other variables in the call frame which must also be copied. The output frame is used in the unification of call arguments with the arguments in the clause head. The unification rules are given below.

2. On commit, those variables in the output frame that were assigned values during unification must be copied back to the call frame. The local frame will also become public. Because of the constraint in Parlog that only one (parallel) goal can produce the value of a shared variable in the calling frame it is unnecessary to check on copy back whether any variable has been instantiated by some other goal since a copy of it was made.

3. When the clause body calls and their descendants have completed, the local frame can be compacted by retaining only those variables in the local frame accessible in the rest of the environment. Because of the unification algorithm only those local variables referenced by the call frame are accessible to the rest of the environment. This stage is optional but considered a vital optimisation as long chains of references can be shortened. A garbage collector process could do this in parallel with the main execution.

4.2. Unification Rules

The matching of terms with terms is the same as in conventional unification. The differences are in the way variables are assigned values. The following rules apply (all variables are dereferenced):

1. Unifying a simple term to an uninstantiated variable: in this case the term is simply assigned to the value cell of the variable.

2. Unifying two uninstantated variables: the least recent one (which maybe in the output frame) is assigned a pointer to the other (which is always in the local frame).

Exception: if the local argument is an input argument then a pointer can be assigned to the variable in the call frame.

3. Unifying a complex term to a call variable: this case fits naturally into this scheme; using structure sharing, the variable is assigned a <skeleton,frame> pair where the skeleton pointer points to the structure in the code area (static memory) and the frame pointer is the local frame, where the values of any variables in the term are kept.

4. Unifying a complex term to a local variable: this involves extra copying to ensure the constraint that variables can only dereference to variables in the local frame. The local variable is assigned a <skeleton,subframe> pair where the subframe is a frame created in the local frame to store value cells of any variables in the complex term. The call variables are assigned pointers to the variables in this subframe.

Exception: if the local argument is an input argument then a <skeleton,frame> pair is assigned to the local variable where the frame is the call frame.

Figure 2 shows an example of the bindings resulting from this unification.

Input variables can be exempted from this rule because they can

Unify the call g(a,b,c,[d|e]), where 'a' is instantiated to 5, with the clause head g(w,x,[x|y],z)

call frame	f0	->	a : 5 b : undef c : undef d : undef e : undef
output frame	f0	->	b : x.f1 c : [x|y].f1 d : d.f1a e : e.f1a
local frame	f1	->	w : 5 x : undef y : undef z : [d|e].f1a
	f1a	->	d : undef e : undef

Figure 2: An example using the unification rules.

not update ancestor frames and thus it is 'safe' to access these shared frames directly.

This means that a variable shared between two or more goals in a clause can only be updated by its producer; consumer goals will dereference the variable to a term in the frame of the producer or one of its descendants.

This also reduces the overhead of unifying complex terms with local variables (case 4) since most complex terms used in this way are input terms. Output terms are usually constructed by unifying complex terms in the local clause with a call variable (case 3).

The strict mode declarations of Parlog mean that the copying of variables from call to output frames is not necessary. Input arguments can be accessed directly from the call frame; output arguments can be assumed to be undefined and on commit their values copied back to the corresponding (dereferenced) variables in the call frame. These optimisations are not possible in Concurrent Prolog except with read-only variables.

4.3. Frames.

A clause requires access to three frames for unification:

The call frame
to access input variables (this is read-only during unification).

The output frame
to store (private) bindings of output variables.

The local frame
to store all variables in this clause.

Pointers to these frames are kept in the process control block of the and-processes executing the clause. The call instruction has to supply the call frame pointer.

If each processor has some local memory then the output and local frames could be stored here until unification succeeded. Then the neck instruction would allocate environment frames in dynamic memory in which these are stored.

If a process suspends during unification and is removed from its processor then the contents of the local memory must be saved in a 'swap' block in dynamic memory so that any processor can reactivate the process. This pointer is also stored in the process control block.

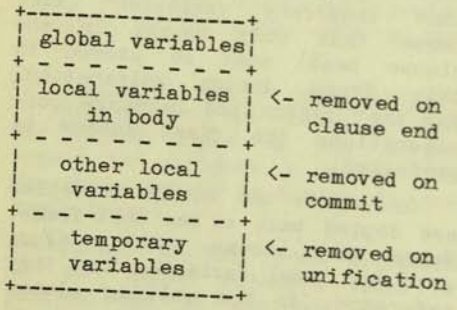
An alternative to this scheme is to precede unification instructions with an init instruction to allocate frames in dynamic memory, replacing the neck instruction. This has the advantage that swap blocks are not needed but the disadvantage that a frame must be allocated for processes that subsequently fail unification and that temporary variables (i.e. those that only appear in the clause head) must be stored in this frame during unification. For that reason and hardware considerations the first scheme is preferred.

On commit the output variables are copied back to the call frame. Hence they become public, along with any local variables that they reference, to the calling clause and its other descendants.

If there is no guard, i.e. the clause commits immediately after successful unification then the neck and commit instructions can be replaced by an ncommit instruction that writes the local output variables immediately to the call frame and allocates a frame only for local variables in dynamic memory.

Because of the way in which the control structure has been separated from the environment structure, the third stage of unification cannot be done automatically on the termination of the clause. Instead, each environment frame contains a count of the number of processes accessing this frame and a pointer to its call frame. When the reference count drops to zero (i.e. the clause has been completed) the frame can be merged with its call frame. Variables in the local frame that are referenced by call variables are checked to see if they have been assigned a simple term. If so this term is assigned to the variable in the call frame. Then only the variables in the local frame which are still referenced by the call frame need to be saved.

To ease the compaction of a frame, the variables can be stored so that variables which can always be removed first are stored after the other variables i.e.:



Snapshots of the environment are given in Figure 3 for the call partition(5,[3,1,7,2],x1,x2). The relation partition is defined as:

```
mode partition (?,?,^,^)
partition(_,[],[],[]) .                    (1)
partition(u,[v|x],[v|y],z) :-              (2)
      v<u | partition(u,x,y,z) .
partition(u,[v|x],y,[v|z]) :-              (3)
      v>=u | partition(u,x,y,z)
```

```
f0  ->  x1: [v|y].f1'
        x2: z.f1'

f1' ->  v : 3
        y : [v|y].f2'
        z : z.f2'
        u : 5
        x : [1,7,2].-

f2' ->  v : 1
        y : y.f3"
        z : [v|z].f3"
        u : 5
        x : [7,2].-

f3" ->  v : 7
        y : [v|y].f4'
        z : z.f4'
        u : 5
        x : [2].-

f4' ->  v : 2
        y : [].-
        z : [].-
        u : 5
        x : [].-
```

Figure 3a: state of environment after the terminating case (before compaction).

```
f0  ->  x1: [v|y].f1'
        x2: z.f1'

f1' ->  v : 3
        y : [v|y].f2'
        z : [v|z].f3"
        u : 5
        x : [1,7,2].-

f2' ->  v : 1
        y : y.f4'

f3" ->  v : 7
        y :
        z : [].-

f4' ->  v : 2
        y : [].-
```

Figure 3b: intermediate stage in compaction (compacted to frame f1').

```
f0  ->  x1: [v|y].f1'
        x2: [v|z].f3"

f1' ->  v : 3
        y : [v|y].f2'

f2' ->  v : 1
        y : y.f4'

f3" ->  v : 7
        y :
        z : [].-

f4' ->  v : 2
        y : [].-
```

Figure 3c: environment after compaction.

Notice that structure sharing means that garbage collection is not optimal. For example, in frame f3", figure 3c: because z is the third variable in the frame, y (which is the second) is implicitly stored even though it is not referred to (Bruynooghe 1982).

5. Discussion

5.1. Design issues. Separating the control structure from the environment structure incurs some overhead in that two explicit trees have to be maintained. In particular, the environment frames contain explicit links between them (from child to parent) to represent the environment tree, and the environments have to maintain a count of the processes attached to them as this is can not be deduced from the control tree.

However, by removing processes which are only waiting for children to terminate, as in the case of a committed clause waiting for body goals, we reduce the number of suspended processes in the system, thus taking advantage of the determinism that the commit offers. Also when failure occurs in some body goal, the failure is immediately effected at the top level of goal which it affects (which may be a guard or a top level query) and does not have to be transmitted up the control tree.

Localising unification to three frames (call input, call output and local) has a number of advantages:

a) The amount of copying done on unification and commit is very small, particularly since the mode of every variable is determined at compile time.

b) Variables can be referenced directly by their address (once in dynamic memory) rather than by a <variable,frame no.> pair where the frame number is different for each different instance of a variable (i.e. there is no problem of multiple instances of the same variable).

c) The output and local frames can be stored in high speed local memory during unification. Each processor could have unification hardware to carry out the unification instructions in this memory if unification proves to occupy a significant fraction of the system's effort.

5.2. Current work.

A Parlog compiler into abstract machine code and a simulator of the multi-processor machine, are being written (in C on Unix) to implement all the features of Parlog. This will eventually include those features not mentioned in this paper (e.g. set-expressions) Eventually, the development of special-purpose hardware is envisaged. However, it would clearly be premature to embark on this prior to a detailed study of the performance of the simulated system.

It would appear from this design that the overhead of process creation and termination and management of environment frames are very critical to the performance of the system. As well as attempting to minimise these overheads (perhaps using special hardware) we will be studying the effect of these overheads to determine what degree of parallelism is necessary before parallel programs run faster on a multiprocessor system than sequential Prolog on a single processor.

REFERENCES

M. Bruynooghe, "Memory Management of Prolog Implementations", pp. 83-98 in Logic Programming, ed. S.-A. Tarnlund, (1981).

M. Bruynooghe, "A Note on Garbage Collection in Prolog Interpreters", pp. 52-55 in Proc. of the First International Logic Programming Conference, ed. M. van Caneghem, ADDP-GIA, Faculte des Sciences de Luminy, Marseille. (September 1982).

K. Clark and S. Gregory, "A Relational Language for Parallel Programming", pp. 171-178 in Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture, (1981).

K.L. Clark and S. Gregory, "Parlog: Parallel Programming in Logic", Dept. of Computing Research Report DOC 84/4, Imperial College, University of London (April 1984).

K.L. Clark and S. Gregory, "Parlog: A Parallel Logic Programming Language", Dept. of Computing Research Report DOC 83/5, Imperial College, University of London (March 1983). (Draft)

J.S. Conery and D.F. Kibler, "Parallel Interpretation of Logic Programs", in Proc. of the ACM Conference on Functional Programming Languages and Computer Architecture, (1981).

K. Furukawa, K. Nitta, and Y. Matsumoto, "Prolog Interpreter Based on Concurrent Programming", pp. 38-44 in Proc. of the First International Logic Programming Conference, ed. M. van Caneghem, ADDP-GIA, Faculte des Sciences de Luminy, Marseille. (September 1982).

S. Haridi and A. Ciepielewski, "An Or-Parallel Token Machine", pp. 537-552 in Proc. of the Logic Programming Workshop '83, Nucleo de Inteligencia Artifical, Universidade Nova de Lisboa (July 1983).

S. Kasif, M. Kohli, and J. Minker, "PRISM - A Parallel Inference System for Problem Solving", pp. 123-142 in Proc. of the Logic Programming Workshop '83, Nucleo de Inteligencia Artifical, Universidade Nova de Lisboa (July 1983).

E. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter", Report CS83-06, Dept. Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel (February 1983).

P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", ACM Computing Surveys, Vol. 14, (1) pp. 93-143 (March 1982).

D.H.D. Warren, "Implementing Prolog - Compiling Predicate Logic Programs", DAI Research Report no.39 & 40, Edinburgh University (1977).

A HIGHLY PARALLEL PROLOG INTERPRETER BASED ON THE GENERALIZED DATA FLOW MODEL

Peter Kacsuk
Institute for Co-ordination
of Computer Techniques
1015 Budapest, Donáti u 35-45.
Hungary

ABSTRACT

A generalized data flow model and its applications for constructing a highly parallel Prolog interpreter is described in this paper. The parallel Prolog interpreter is suited to utilize advantages of OR-and AND-parallelism as well. Transformation of the AND-OR tree into a data flow graph based on the Generalized Data Flow Model is shown. Operator types needed for parallel evaluation of Prolog programs are explained in detail.

1 INTRODUCTION

Recently Prolog has gained an increasing popularity in such areas of computer techniques as artifical intelligence, expert systems, and so on. Practical applications of Prolog need an effective implementation of the language, including fast execution of programs inherently containing a lot of backtracking steps. Nowadays several research projects have started aiming at the solution of parallel interpretation of Prolog programs. The majority of these research directions fundamentally support the so-called AND/OR process modell /Conery and Kibler 1981/ or /Eisinger, Kasif and Minker 1982/. A parallel Prolog interpreter based on this model creates dynamically AND-processes and OR-processes in branching points of the AND-OR tree and allocates these processes to idle processors being in the system.

Recently several data flow models have been proposed for parallel execution of Prolog programs /Moto-oka and Fuchi 1983/ and /Umeyama and Tamura 1983/, but these models have been mapped into conventional multiprocessor systems.

A new method for parallel implementation of a Prolog interpreter is presented in this paper. The essence of the method can be shortly described as follows:

First the AND-OR tree of Prolog program is transformed into a data flow graph based on the Generalized Data Flow Model /Kacsuk 1983a/. The next step is to map this data flow graph into a regular, homogenous processor space in which each processor can communicate only with its neighbours. In this way both the interpreter program and the Prolog program with its data base will be distributed in the processor space. Loading of Prolog programs into the processor space is executed during the compilation.

After loading, processors can work in parallel and they are activated by the firing rule of the Generalized Data Flow Model. The data flow execution mechanism of interpreter assures the logical exploitation of inherent AND/OR parallelism, meanwile the regular processor space gives the possibility of physically parallel execution of Prolog programs.

Chapter 2 is a short description of the Generalized Data Flow Model - the theoretical background of the parallel Prolog interpretation mechanism. Chapter 3 applies the Generalized Data Flow Model to realize a parallel interpreter that exploits OR-parallelism. Operator types playing an important role in utilizing OR-parallelism are thoroughly described. In chapter 4 the solution of realizing AND-parallelism is carefully investigated.

2 THE GENERALIZED DATA FLOW MODEL /GDM/

Although the pure data flow model is well suited for exploiting inherent parallelism in functional programs, it can not be directly applied for parallel interpretation of Prolog programs. The most important reason is that operators should have an inner state for backtracking since the unidirectional data flow graph is inadequate to describe the backtracking behaviour of Prolog interpreter.

The Generalized Data Flow Model /GDM/ originally intended for programming multiple microprocessor systems can be applied for realizing the backtracking mechanism of Prolog interpreter. A detailed description of GDM can be found in /Kacsuk 1983a/. This paper just summarizes features of GDM inevitable for understanding parallel interpretation of Prolog programs:

1. Functions associated with nodes can be unlimitedly complicated and there is no limitation to the number of input and output arcs of nodes.

2. One node can be associated with several functions. A subset of these functions is activated by a firing situation. This way one operator can simultaneously product a lot of results and these can be sent to different subsets of output arcs.

3. Functions associated with nodes are evaluated on the basis of conventional Neumann-style control flow semantics, so they can contain temporary variables /local memories/.

4. Operators can have an inner state playing role in:

- selecting the input arcs on which tokens are needed for firing,
- determining the new inner state created by the firing,
- selecting operator functions to be executed during the current firing,
- selecting the subset of output arcs responsible for sending result tokens of the current firing.

5. Operators can preserve partial results from one firing for a later one.

6. There are no restrictions for data structures carried by tokens.

Mapping a data flow graph based on GDM into a homogenous, regular processor space /often called cellular space/ each operator is associated with a cell that can be represented by a generalived transition function /f/:

f /state, {inp-tok} , {action}/ = /next-state, {out-tok} /

where the inner state of the cell before firing is "state" and after firing is "next-state". {inp-tok} means the set of input arcs and input tokens taking part in the firing of the operator. {action} means the set of procedures to be executed during the fire. {out-tok} means the set of output tokens produced by the firing.

3 THE DATA FLOW SEARCH TREE /DST/

Executing a Prolog program is equivalent with searching in an AND-OR tree reflecting the structure of the Prolog program. As an example for constructing the AND-OR tree consider the following simple Prolog program:

```
a (X,Y) : b(X,Y), c(X),d (Y ).
b (orange, apple).
b (orange, lemon).
b (plum, apple).
c (orange).
d (lemon).
```

The structure of AND-OR tree belonging to the above described Prolog program is shown in figure 1. depicting with circles the clause-heads /procedures/ and with rectangles the procedure calls /atomic goals/.

The AND-OR tree well demonstrates the control mechanism of the sequential Prolog interpreter. Before entering a procedure body a unification step is needed. Entering is allowed only when unification of the actual and formal parameters has been successful. Variable bindings created during unification must be passed to atomic goals in the clause body. The sequence of these two actions /unification and parameter passing/ is denoted by dashed lines in the circles representing rules /van Emden 1982/. Unit clauses have no body therefore in circles representing unit clauses dashed lines are missing.

Dashed arcs in the AND-OR tree represent the successful return from a procedure. After a failed unification the interpreter activates the backtracking mechanism. If the current literal has another OR-branch right to the failed OR-branch then the interpreter tries a new unification selecting the next clause.

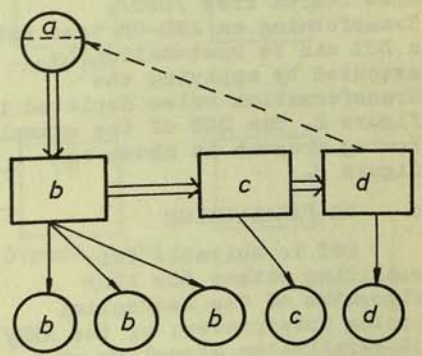


figure 1.

When all the OR-branches deriving from the current procedure call have sent back a failure signal, then the interpreter backtracks along the failed atomic goal.

After all one can say that the AND-OR tree represents the control flow graph of the Prolog interpreter, where the direction of arcs shows the progress of the Prolog program and the opposite direction serves for controlling the backtracking mechanism.

On the basis of the Generalized Data Flow Model AND-OR trees can be transformed into data flow graphs that can contain 4 types of operators:

1. UN /or UNIFY/
2. AND /or BODY/
3. OR /or CALL/
4. UT /or UNIT/

In the data flow graph arcs connecting nodes represent data pathes instead of control pathes. Accordingly between two nodes there are two arcs with opposite directions, one for passing the actual parameters and the other for sending back results. The data flow graph derived from the AND-OR tree will be named Data Flow Search Tree /DST/. Transforming an AND-OR tree into a DST can be systematically executed by applying the transformation rules depicted in figure 2. The DST of the example Prolog program is shown in figure 3.

4 OR-PARALLELISM

DST is suitable for realizing either the LRDF algorithm of the sequential Prolog interpreter, or the AND/OR-parallelism algorithm of a parallel Prolog interpreter. The realized algorithm depends on determining the functions of operators applied in DST and the types of tokens moving on the arcs of DST.

First, function of operators and types of tokens will be determined for utilizing the inherent OR-parallelism of Prolog programs. The general structure of a token is the following:

- token type
- context field
- data field

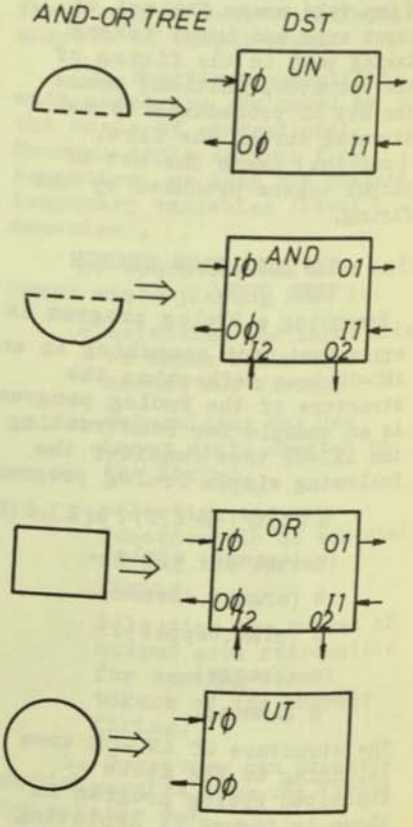


figure 2.

Token types needed for OR-parallelism are the following:

- DO : first call of a procedure
- REDO : repeated call of a procedure
- SUCC : successful return from a procedure
- FAIL : failed return from a procedure

The context field serves for distinguishing tokens originating from distinct instances of the same procedure call e.g. due to recursion. The length of data field can be variable, it contains the actual parameters in DO tokens and results in SUCC tokens. The data field is missing in REDO and FAIL tokens.

Now a short description of operator types realizing the control mechanism of a parallel Prolog interpreter based on the DST follows.

4.1 UNIFY-operator /UN/

The transition function of UN-operator is the following:

f(idle,DO(IØ),UNIFY(SUCCESS))=
=(wait,DO(O1)) (1)

f(idle,DO(IØ),UNIFY(FAILED))=
=(idle,FAIL(OØ)) (2)

f(wait,SUCC(I1),-)=
=(idle,SUCC(OØ)) (3)

f(wait,FAIL(I1),-)=
=(idle,FAIL(OØ)) (4)

f(idle,REDO(IØ),-)=
=(wait,REDO(O1)) (5)

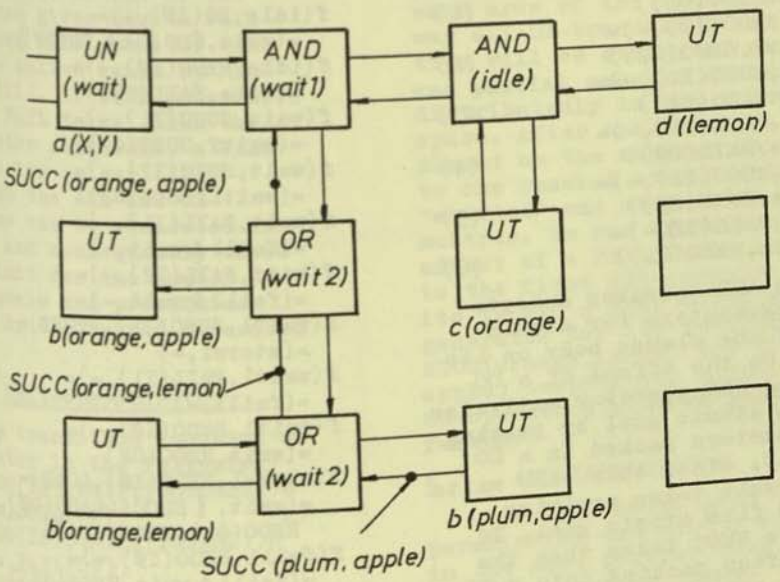


figure 3.

The UN-operator has two tasks. First it makes unification among locally stored formal parameters and actual parameters packed into DO token arriving on its IØ input arc.

Secondly after a successful unification UN passes parameters with new variable bindings to the clause body on its O1 output arc (1). In the case of a failed unification UN sends back a FAIL token to the caller (2). After evaluating the clause body the result token arriving on the I1 arc is sent back to the caller (3), (4). REDO tokens arriving on the IØ arc are passed without change to clause body (5).

4.2 AND-operator

The transition function of AND-operator is the following:

f(idle,DO(IØ),-)=
=(wait1,DO(O1)) (1)
f(idle,REDO(IØ),-)=
=(wait2,REDO(O2)) (2)
f(wait1,SUCC(I1),-)=
=(wait2,DO(O2)) (3)
f(wait1,FAIL(I1),-)=
=(idle,FAIL(OØ)) (4)
f(wait2,SUCC(I2),-)=
=(idle,SUCC(OØ)) (5)
f(wait2,FAIL(I2),-)=
=(wait1,REDO(O1)) (6)

The AND-operator accepts actual parameters for atomic goals of the clause body on its IØ arc. On the effect of a DO token the AND-operator activates the firs atomic goal by passing the parameters packed in a DO token (1). After this, AND waits for a result token arriving from the first atomic goal. If this is a SUCC token then the result after packing into a DO token is passed to the second atomic goal (3). Also, the second atomic goal is activated when the AND-operator receives a REDO token on its Ø arc, but the activation token is of REDO type (2). The result of the second procedure call arrives on the I2 arc. If this is a SUCC token then the result is send back to the caller operator (5). This result contains the variable bindings created by the first and second literal of the AND-operator.

When the first atomic goal sends back a FAIL token, then the AND-operator immediately sends back this FAIL token to the caller (4). When the second atomic goal gives back a FAIL token a backtrack is needed. This is executed by sending a REDO token to the first atomic goal (6).

4.3 OR-operator

The transition function of OR-operator is the following:

f(idle,DO(IØ),-)=
=(wait,{DO(O1),DO(O2)}) (1)
f(idle,REDO(IØ),-)=
=(idle,FAIL(OØ)) (2)
f(wait,SUCC(I1),-)=
=(wait2,SUCC(OØ)) (3)
f(wait,SUCC(I2),-)=
=(wait1,SUCC(OØ) (4)
f(wait,FAIL(I1),-)=
=(fail1 & wait,-) (5)
f(wait,FAIL(I2),-)=
=(fail2 & wait,-) (6)
f(wait1,SUCC(I1),STORE)=
=(store1,-) (7)
f(wait1,FAIL(I1),-)=
=(fail1,-) (8)
f(wait1,REDO(IØ),-)=
=(wait,REDO(O2) (9)
f(store1,REDO(IØ),LOAD)=
=(wait, { SUCC(LOAD)(OØ),
REDO(O1),REDO(O2)}) (10)
f(fail1,REDO(IØ),-)=
=(fail1 & wait,REDO(O2)) (11)
f(fail1 & wait,SUCC(I2),-)=
=(fail1,SUCC(OØ)) (12)
f(fail1 & wait,FAIL(I2),-)=
=(idle,FAIL(OØ)) (13)

The OR-operator accepts actual parameters packed into DO token on its IØ arc. The OR-parallelism is realized by the fact that the OR-operator simultaneously passes actual parameters to the two connected procedures (1). These two procedures produce result tokens in contest with each other. The result arrived in the firstly received SUCC token will be sent back to the caller by the OR-operator (3) or (4). The result of the secondly received SUCC token will be stored locally within the OR-operator by the STORE function (7). When the OR-operator gets a new REDO token the second result will be fetched by a LOAD function and sent back immediately to the caller while REDO tokens will be sent to the two connected procedures for producing new results (10). If one of the procedures gives back a FAIL token then this operator will never be called again by REDO tokens (11). If both procedures produce FAIL token, then the OR-operator also sends a FAIL token to its caller (13).

Both the AND-and OR-operators can be connected in cascade and consequently there is no limit for the number of atomic goals and procedures handled by the method described above.

4.4 UNIT-operator (UT)

The transition function of UT-operator is the following:

f(idle,DO(IØ),UNIFY(SUCCESS))=
=(idle,SUCC(OØ)) (1)

f(idle,DO(IØ),UNIFY(FAILED))=
=(idle,FAIL(OØ)) (2)

f(idle,REDO(IØ),-)=
=(idle,FAIL(OØ)) (3)

The UT-operator unifies input parameters packed into DO token with the locally stored part of the data base. If the unification is successful parameters with new variable bindings are sent back in a SUCC token to the caller by the UT-operator (1). When the unification failed or UT has got a REDO token, then a FAIL token is sent back by the UT-operator (2) or (3).

As an example to understand how operator types introduced up-to-now work together let's consider figure 2, that shows a snapshot of tokens moving in the DST. OR-operators are in wait2 state waiting for results from the second and third b clauses. The first AND-operator is in waitl state waiting for result from its first atomic goal. There are 3 tokens in the DST, since the OR-operators representing an OR-branch in the AND-OR tree produces tokens for each arcs of the branch. This way all OR-branches of an AND-OR tree will be evaluated paralleIy and partial results are stored distributedly in the processor space. After backtracking tokens placed on the subtree belonging to the reasked atomic goal are "shifted" one step ahead in the subtree. In our example on the effect of a FAIL token arriving to the first AND-operator on its I2 arc a REDO token will be generated for its subtree so SUCC(orange,lemon) and SUCC(plum, apple) temporarily stored in OR-operators will be passed further.

5 AND-PARALLELISM

In this model AND-parallelism can not be utilized in the most general case when the direction of formal parameters is not known before execution. Under such circumstances only OR-parallelism can be exploited as described in

chapter 4. However in many cases programers know in advance during writing their program that parameters of certain procedures can be only input or only output parameters. For example in DEC-10 Prolog system programer can fix direction of parameters by means of the mode declaration. For procedures supplied with mode declaration compiler can generate a data dependency graph (Conery and Kibler 1983), that shows which atomic goals of the procedure body can be evaluated in parallel. For example consider the following procedure:

a(X,Y) : b(X,Z), c(X), d(X,Y,Z), e(Y), f(X,Z).

Data dependency graphs of the procedure for two different mode declarations are shown in figure 4.

An atomic goal is named consumer of an X variable, if the arc representing X in the data dependency graph is the input arc of the node representing this atomic goal. Similarly an atomic goal is producer of X, if the X arc is the output arc of the atomic goal node. A procedure call is on the i-th level, if every variables for which this procedure call is a consumer were produced on any of the Ø, 1, ..., /i-1/ levels. Øth level belongs to the head of the procedure.

On the basis of the AND-OR tree and data dependency graphs the compiler can generate DST well suited for realizing AND-parallelism as well. For this purpose DST operator types introduced up-to-now are extended and a new token type /VALID/ is introduced. Extended operator types are the following:

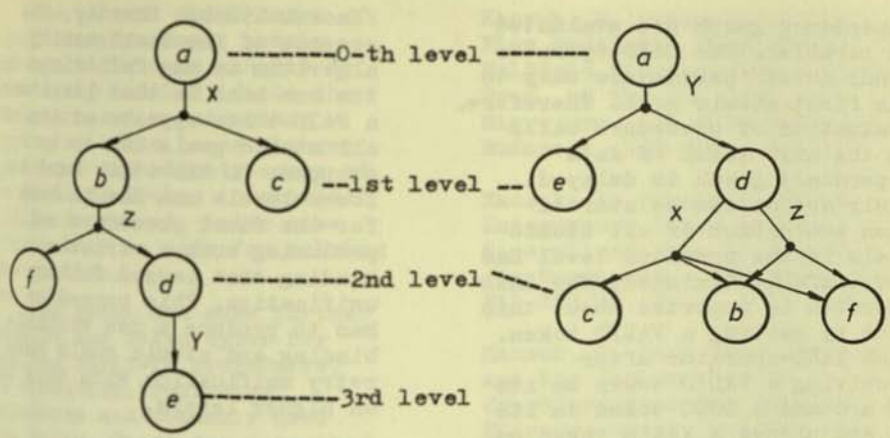
original operators	extended operators
UN	PUN /Par. UNIFY/
AND	PAND /Par. AND/
AND	SAND /Seq. AND/
OR	POR /Par. OR/
UT	PUT /Par. UNIT/

Extended operators except PAND and SAND are different from their original version only in handling VALID tokens. PUN sends a VALID token after having sent a DO or REDO token on its O1 arc. PUT and POR are able to receive a VALID token, but they have no action defined for a VALID token.

Pand and SAND are different from the AND-operator in some aspects. To understand their action consider figure 5 showing DST of the example procedure with two different mode declarations. For constructing DST one has to start from the data dependency graph of the procedure. Since atomic goals on the same level can be evaluated in parallel they are connected with each other by PAND-operators in DST. On the other hand atomic goal groups on different levels must be evaluated sequentially so they are separated from each other by SAND-operators in DST.

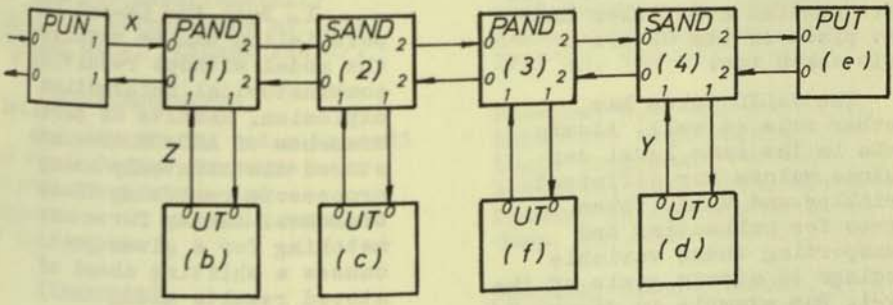
To go further, an informal description of PAND-and SAND-operators is given. For atomic goals, variables are divided into two classes: produced or consumed by the atomic goal. This information is given by the compiler on the basis of data dependency graph.

A PAND-operator is activated by a DO token coming on IØ arc. Actual parameters packed into a DO token are passed simultaneously to atomic goals connected to the PAND-operator. In this way procedure calls on the same level of the data

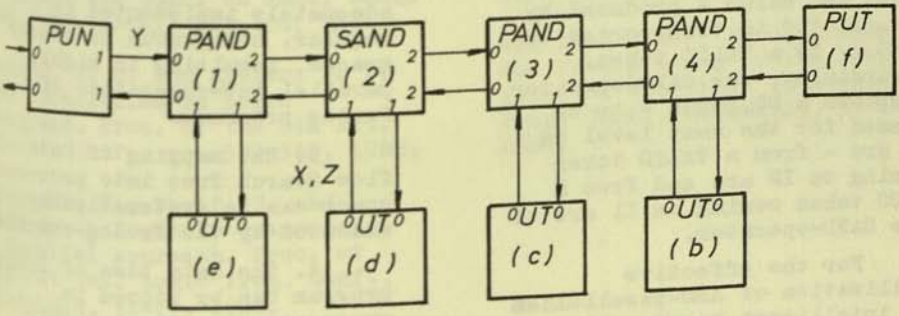


/a/ mode a/+,-/ /b/ mode a/-,+/

figure 4. Data dependency graphs for the example procedure



/a/ mode a/+,-/



/b/ mode a/-,+/

figure 5. DST for the example procedure

dependency graph are evaluated in parallel. The SAND-operator sends actual parameters only to the first atomic goal. Therefore, evaluation of procedure calls in the next level of data dependency graph is delayed. Their action can be started when evaluation of all atomic goals in the previous level has successfully finished. The SAND-operator is reported about this fact by getting a VALID token. Each PAND-operator after receiving a VALID token on its IØ arc and a SUCC token on its I1 arc places a VALID token on its O2 arc. The SAND-operator sends a DO token on its O2 arc when it has got a SUCC token from its first atomic goal and a VALID token from his caller. After sending a DO token SAND must place on its O2 arc a VALID token too.

The VALID token has another role as well. Atomic goals in the same level can produce values for different variables and VALID token serves for collecting and transporting these variable bindings to atomic goals of the level. For example in figure 5/a value X produced by procedure head /PUN/ is transported to the SAND/2/-operator by a DO token and value Z produced by literal "b" is transported to SAND/2/ by a VALID token. Accordingly, the SAND-operator composes a DO token - to be passed for the next level on O2 arc - from a VALID token coming on IØ arc and from a SUCC token coming on I1 arc of the SAND-operator.

For the effective utilization of AND-parallelism an intelligent backtracking algorithm - like on /Pereira and Porto 1980/ was planned in /Kacsuk 1983b/. Shortly, the essence of the backtracking algorithm is the following: On the one hand in that level where a FAIL token appears action of all atomic goals has to be stopped. On the other hand in lower levels one has to look for the first procedure call producing such a variable binding that caused failure of unification. This procedure call has to produce a new variable binding and atomic goals have to retry unification with this one on higher levels.

CONCLUSION

The proposed parallel Prolog interpreter has the following advantages:

1. Both the OR-and AND-parallelism can be realized in the model without resulting a combinatorical information explosion. Results of parallel branches of AND-OR tree are stored distributedly among processors realizing those branches. Asking for a new matching for a given goal causes a shifting ahead of stored results along the processors. The model can assure intelligent backtracking.

2. The model can be adequately implemented in regular, homogenous processor spaces, resulting in highly parallel interpretation of Prolog programs.

3. The mapping of Data flow Search Tree into processor space can be systematically executed by the Prolog compiler.

4. The data base of Prolog program can be stored in processor space in a highly distributed way.

The present model has a significant drawback, namely big data structures can not be effectively handled. Since the model is based on structure - copying as lists become longer and longer the effectiveness of the parallel Prolog interpreter will be decreased. Later some efforts must be made for solving this problem.

The model has some further operator and token types for handling built-in procedures, CUT operation, recursive procedures and commonly used data bases. These are described in /Kacsuk 1983b/. In the present phase of the research a simulator based on the Generalized Data flow Model has been constructed and used for experimentally justifying correctness ot this proposed model /Kacsuk 1983c/.

ACKNOWLEDGEMENTS

Many thanks to Peter Szeredi and Peter Garami for some very helpful discussions and suggestions to an early version of this paper.

REFERENCES

Conery, J. S. and Kibler, D. F. Parallel interpretation of Logic Program. ACM Conf. on Func. Lang. and Comp. Arch., 163-170, 1981.

Conery, J. S. and Kibler, D. F. And Parallelism in Logic Programs. Proc. of the 8th Art. Intell. Conf., pp. 539-543, 1983.

Eisinger, N., Kasif, S. and Minker, J. Logic Programming: A Parallel Approach. Proc. of the 1st Int. Logic Prog. Conf., Marseille, 71-77, 1982.

Kacsuk, P. Generalized Data Flow Model for Programming Multiple Microprocessor Systems. Proc. of the 3th Symp. on Microcomp. and Microproc. Appl., Budapest, 539-551, 1983./a/

Kacsuk, P. Parallel Prolog Interpreter Based on the Generalized Data Flow Model. /in Hungarian/, Technical Report ELL-259/83, SzKI, 1-51, 1983./b/

Kacsuk, P. Data Flow Simulator and its Implementation Considerations for Multiple Microprocessor Systems. /in Hungarian/, PhD thesis, University of Technology Budapest, 1-132, 1983 /c/

Moto-oka, T. and Fuchi, K. The architectures in the fifth generation computers. Proc. of IFIP'83, 589-602, 1983.

Pereira, L.M. and Porto, A. Selective backtracking for logic programs. 5th Conf. on Automated Deduction, Les Arcs; France, in Lecture Notes in Comp. Sci. 87, 306-317, 1980.

Umeyama, S. and Tamura, K. A parallel execution model of logic programs. Proc. of 10th Int. Symp. on Comp. Arch., 349-355, 1983.

van Emden, M.H. An Algorithm. for Interpreting Prolog Programs. Proc. of the First Int.Logic Prog. Conf., Marseille, 56-64, 1982.

UNIFICATION FOR A PROLOG DATA BASE MACHINE

G. Berger Sabbatel, W. Dang, J.C. Ianeselli, G.T. Nguyen+
IMAG/TIM3 Equipe d'architecture des calculateurs
INPG - 46 av. F. Viallet - 38041 GRENOBLE CEDEX
+ IMAG - Laboratoire de génie informatique
B.P. 68 - 38402 St MARTIN D'HERES - FRANCE

ABSTRACT

This paper addresses the problem of the unification in the context of a Prolog database machine based on a multiprocessor architecture, with parallel access to a set of disks (OPALE project). A search strategy based on parallelism and set processing is briefly exposed, and the architecture of OPALE is outlined.

A decomposition of the unification is proposed. A part of it (the preunification) can be executed by a hardware operator at the disk transfer rate. It allows a significant selection of the data, so that the whole unification can be completed on the fly on a 16 bits microprocessor. The hardware architecture of the search operator is presented.

1. INTRODUCTION:

During the past few years, a lot of work has been done in the field of data base machines, aiming at improving the performances of relationnal data bases. A number of tools have been defined, such as sequential filtering operators, parallel architecture and algorithms, etc... (Bancilhon, Richard and Scholl 1981, De Witt 1979, Gardarin 1981).

On the other hand, the development of artificial intelligence have led to new requirements for data bases, such as deduction, knowledge processing, etc... Logical programming (and PROLOG (Roussel 1975)) often apears as a promising approach (Gallaire 1981, Warren 1981).

With the development of technology (VLSI, secondary memories), it seems interesting to study new models for data bases along with architectures suited for their implementation. Hence, the OPALE project (Berger Sabbatel, Ianeselli and Nguyen 1983) aims at designing a data base machine oriented toward logical programming.

In the context of this project, the unification appears as the kernel of the interpretation problem. In this paper, we will first outline the project, and give the architecture of the OPALE machine, based on a multiprocessor structure, and execution of the unification through a hardware operator. We will then focus on the unification, and give a decomposition of it which allows the "on the fly" unification of sets of goals with sets of clause headers read from a disk.

2. PROLOG AND DATA BASES:

2.1. Introduction:

Two ways of using PROLOG in data bases can be considered: interfacing it with relationnal data bases (Chakravarthy, Minker and Tran 1982), or use it directly as a data base using clauses for representing data (relations).

Every relationnal operator can be expressed in PROLOG, and several operators can be combined in simple clauses. Furthermore, PROLOG can be considered as a superset of relationnal algebra, as additionnal features exists, such as manipulation of implicitly defined relations, processing of non atomic data, simple insertion of semantic actions in the data, etc...

Hence, we consider the direct interpretation of PROLOG as the best solution, as it allows the use of full PROLOG capacities without creating a new level of translation / interpretation between the user and the machine. Our claim is that the most efficient tools implemented for relationnal databases can be used as well for the direct execution of PROLOG database accesses.

In OPALE, the packets of clauses will be stored as linear lists of alternatives and accessed through indexing and filtering. Symbols will be coded through a dictionnary. Every data item is prefixed by a type-byte which allows types such as symbols, characters, integer, reals, strings, variables, etc... For the functionnal symbols, the type will include their arity, so that the structure of the terms can be decoded without access to the dictionnary. Most of data items will the be coded on 5 bytes: 1 type byte, and 4 data bytes.

The use of PROLOG for data base management and the design of a specialized architecture, involve particular choices for the interpretation, as there are important differences with program interpretation: large number of alternatives, high fail ratio for the unification, and in most cases the clauses will be in secondary memory. We assume that the ordering of operations and the order of the results are generaly not meaningfull, that very few hard-wired predicates will be encountered in the data base, and that the complexity of the clauses will be relatively small.

2.2. Search strategy:

Classical left to right and depth first sequential interpretation, checks one solution at a time. In a data base environment, this can be a severe drawback, as the optimization of disk accesses would require to access every alternatives of a clause before the verification of another predicate. Furthermore, we intend to exploit the parallelism in the access of several disk units.

Hence, our search strategy for accessing the PROLOG data base aims at three objectives: exploit the parallelism in PROLOG, optimize the disk accesses, and allow the best use of a hardware implemented unifying operator.

The optimization of the disk accesses lead us to a search strategy based on sets. In effect, each search produces sets of solutions (instantiations) (Chakravarthy Minker and Tran 1982). These instantiations may, in turn, produce sets of goals which can be globally verified

through sequential filtering.

Example 1:

```
C (X,Y) <- C1 (X) & C2 (X,Y).
C (X,Y) <- C1 (X) & C3 (X,Y).
C1 (a) <-.
C1 (b) <-.
C2 (b,c) <-.
C3 (a,e) <-.
  :::
```

In this example the search on C1 returns two solutions: X = a, and X = b, which, in turn produces two goals for the first alternative of C: C2(a,Y) and C2(b,Y). These two goals can be searched in a single disk access on C2, through sequential filtering.

Three types of parallelism will be used. The first type is an OR-parallelism, in which alternatives having a non-empty body can be verified with paralell processes (breadth-first strategy). In the example 1, the request C(a,X)? activates two parallel processes which correspond to the two alternatives of C.

The second type is an AND parallelism. To avoid the problem of interdependancy, we treat it by pipe-lining the verification of the litterals in a clause: A process L is attached to every litteral, it receives the instantiations from the previous litteral, generates the goals, and passes it to a search process C. The C process executes the indexing and disk accesses (Fig. 1.), and return the instantiations to the L process, which passes it to its successor.

The disk latency time imply that several goals may be available when the disk is ready. Hence, the third type of parallelism addresses the unification of sets of goals with the disk data stream. This will be the topic of the section 4.

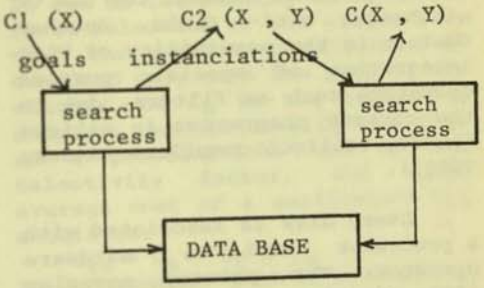


Figure 1.

In conclusion, the parallel search is one of the central issues for high system performances, as it allows an optimal use of the filter. Some additionnal forms of parallelism will also be studied to exploit the multiprocessor architecture of the machine: distribution of search on several disk units, etc...(Conery and Kibbler 1981).

3. ARCHITECTURE:

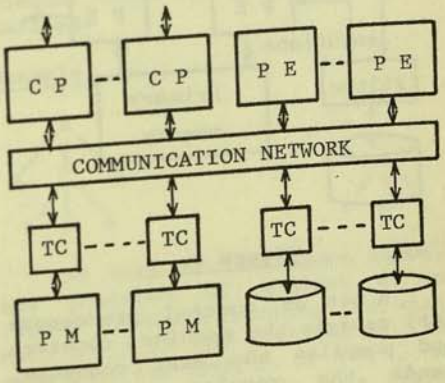


Figure 2.

The architecture of OPALE is depicted on figure 2. It aims at providing fault tolerance and high performances through a distributed

architectures which allows parallel operations on multiple disks. For small or medium sized data-bases, sufficient parallelism can be achieved through the use of winchester mini- disks. Another feature is the possibility of VLSI integration of special- purpose operators such as filters, due to the current progresses in silicon foundry (silicon compilers (Anceau 1983)).

Every disk is associated with a processor and a hardware operator. The primary memories (PM) are also associated with a processor. They manage the sets of environments (intermediate solutions). The processing elements (PE) control the operations.

A verification will then be decomposed in operations on data flows coming from the disks, filtered, processed by the PE and stored in the primary memories. The temporary results will then be used by the PE to generate new goals (Fig. 3).

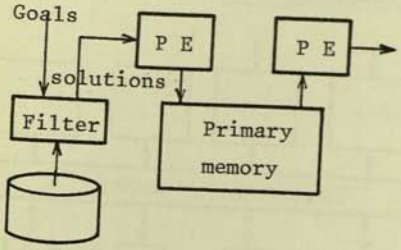


Figure 3.

A set of control processors (CP) manages the machine, receives and compiles the user requests, sends the results. They are connected to the user processors through a local network.

The communication between the various components of the machine will be provided through a message switching network: for a first experiment, we intend to use a multiprocessor parallel bus.

4. UNIFICATION:

4.1. Introduction:

In this section, and in the next one, we will consider the disk processors. They will receive goals, and must find on the disk the clauses whose headers unify with these goals. In order to achieve high performances, our objective is to execute the unification of clause headers with sets of goals "on the fly" in almost all cases, i.e. to process data at the disk transfer rate. When the disk is accessed, our strategy will be to search every clause header which unify with at least one goal in a whole track, in order to minimize the number of disk accesses, and significantly reduce the disk access time (as the major component of access time are the head positionning and rotationnal delays).

We consider that for every packet of clauses accessed on the disk there is a set of goals, even though the indexing can decrease the number of goals per disk access. We also consider that the goals are available to the disk processors as "flat" terms, i.e. that the terms are fully self-contained, without reference to external data (substitutions).

The operation is then to select the clauses whose headers unify with a goal (at least), and then transmit every couple variable/data which describe the instantiations generated by the unification (substitutions) (Fig. 4).

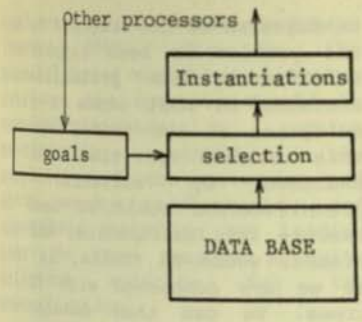


Figure 4.

Some important differences exist between the unification and the selection of data bases:

- the unification operates on tree structured data, and not on normalised relations, furthermore, the data structure may be not statically defined, but defined in the data themselves (typed data).
- the data base can contain variables which may appear not only in the rules headers, but also in facts, where they can stand for irrelevant arguments for example.

In knowledge databases, one may expect that important sets of rules exist, so that we have to allow all these cases to be processed on the fly.

On the other hand, there also exist important differences between the unification in prolog programs and prolog databases:

- large number of alternatives for every clause. The average number of alternatives for a program is generaly less than a tenth. For a data base, it can be of several millions (facts).
- high fail ratio: in PROLOG programs, the fail ratio can be in most cases of about 50%. In databases, one must expect much higher fail ratio, such that 90%, or 99% at least.

These differences involve particular choices for the implementation of the unification. It can be interesting to execute a first selection of the clauses, so that the complete unification is executed only on a few part of the clauses. If C_u is the cost of a complete unification, C_s the cost of a preselection, and Sf the selectivity factor, the total average cost of a unification C_{tu} should be:

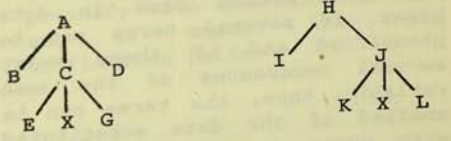
$$C_{tu} = C_s + S_f \times C_u$$

As an example, with a selectivity factor of 10%, the preselection should be usefull if its cost is less than $0.9\ C_u$. This technique should then be very adapted to databases.

4.2. Preunification:

We can define the position of a data item in a term as being the succession of nodes to be accessed from the root to access the element. Every node can be identified by its rank in the arguments of its father node, so that the positions of nodes in unequal trees can be unambiguously defined.

Example:



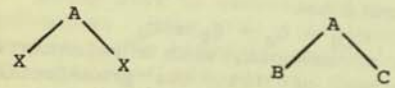
In both of the above terms, the position of element X is (2,2). In the following, we will note pos(N,T) the position of the node N in the term T. By N ∈ T, we mean that N is a node or a leave of the term T.

If two terms are unifiable, then the following condition is verified:

(1) ∀(N1∈T1, N2∈T2 with:
pos (N1,T1) = pos (N2,T2)),
if N1 # N2 then
(N1 is a variable
or N2 is a variable).

The condition is obviously necessary, from Robinson's unification algorithm (Robinson 1965). It is not sufficient:

Example:

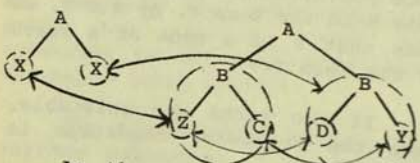


The above terms are obviously not unifiable.

We define the preunification as the operation wich compares two terms, checks the above condition, and produces the list of couples variable-value which constitutes the disagreement of the terms. We name substitutions these couples. The preunification of the terms in the above example produces the couples: X=B, and X=C.

If two terms can be preunified, and if there is only one occurence of every variable in the terms, then, they can be unified. We assume that this is the most common case in data bases. If several terms can be preunified and if there exist several occurences of the same variable, then, the terms can be unified if the data substituted with every occurence of a same variable can be unified.

Example:



As the unification operates on subterms of the original terms, the problem as been simplified. Furthermore, the preunification executes in most cases a first selection of the terms, so that this second operation can be executed by software. The preunification could be used to execute this unification, and the process would be finite, as long as we are concerned with finite trees. We can then demonstrate that the complete unification can be executed with a finite number of preunifications. However, for efficiency reasons, the execution of the second step with a classical algorithm would be more advisable.

In the unification, the substitutions are dynamically applied to the terms, and therefore, checked by the unification operation itself. In the first example, the substitution X=B would have produced the term t(B,B), and the unification fails with the comparison of B with C. It is clear that this modification of the terms involves either a complex data structure to represent the terms, or a complex algorithm which can hardly be implemented on a hardware automaton to be executed on the fly.

Our solution is then to execute "on the fly" the preunification, through a hardware automaton which transmits sets of substitutions for every matched term, and to check the consistency of these substitutions by a program executed on a microprocessor.

4.3. Implementation:

A classical solution for the selection of data in relationnal database machines is the use of

hardware finite state automata (Rohmer 1981). Due to the possibility of variables in Prolog database, the automata should be non-deterministics, which considerably increase the complexity of the automata and their memory requirements, and then their compiling and loading time. We then propose a quite different solution for the preunification.

If a term T can be preunified with (at least) one term among a set of goals C_i , then, the following condition is verified (from the definition of the preunification).

(2)
$\forall$(N $\in$ T, where N is not variable)
if ($\exists$ i, so that
($\exists$ N' $\in$ C_i so that
pos (N',C_i) = pos (N,T))
then N' = N or N' is a variable.

The condition is not sufficient:

Example:

Goals : C1 = t (a,b),
C2 = t (c,d)
Term : T = t (a,d)

The term T meets the condition, but is not preunifiable with one of the goals.

The condition (3) is a negation of the preunification condition:

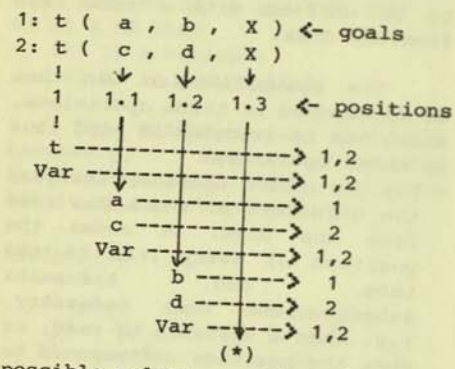
(3)
$\exists$(N $\in$ T, N is not a variable)
$\exists$(N' $\in$ C, N' is not a variable
and pos (N',C) = pos (N,T)
and N' # N)

If (3) is met for a goal C and a term T, then, the term T is not preunifiable with the goal C. The verification of this condition will allow us to eliminate a goal from a set of goals "candidates" to be unified with a term read from the disk.

The preunification can then be decomposed in three operations, which can be executed in pipe line by three operators:

- The structure operator analyses the structure of the terms read from the disk. It codes the position of every item in the term, and transmits substitutions when necessary, i.e. when a variable is read, or when the position corresponds to a variable of a goal.
- The search operator checks the condition 1, by checking for every non variable item read from the disk, if its value is acceptable for its position, i.e. if a goal have the same constant item or a variable for this position. It is then a simple search in a list of values.
- The third operator manages the list of goals which can pre-unify with the term read from the disk (condition (3)). For every item value analysed by the search operator, there is a set of possible goals. The intersection of successive sets gives the set of goals which preunify with the term. If this set is empty, then the unification fails.

Example:



(*) = any value, substitutions generated, no goal substitution.

5. DISK PROCESSORS:

5.1. General description:

The operation of the disk processors is described in the figure 5.

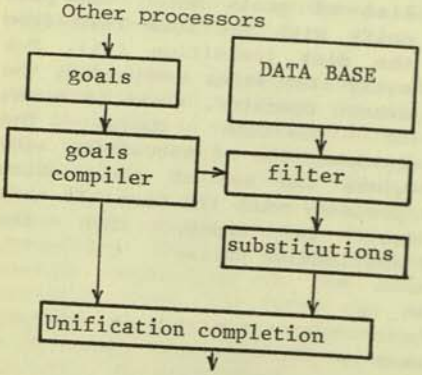


Figure 5.

The goals are received from the other processors of the machine. The goals compiler (software) allows the loading of the memories of the filter. The clauses readout from the database are preunified by the filter (hardware), and then passed to the program which completes the unification, and controls the transmission of the substitutions to the proper process.

The architecture of the disk processors is depicted in figure 6.

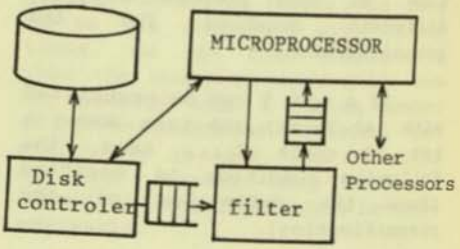


Figure 6.

The microprocessor controls the communication with the other processors of the machine, controls the filter, compiles and loads its programs, and executes the remaining of the unification. It will also control the write operations.

The filter will be composed of 4 parts executing in pipe line the three steps of the preunification and the selection of the substitutions (figure 7).

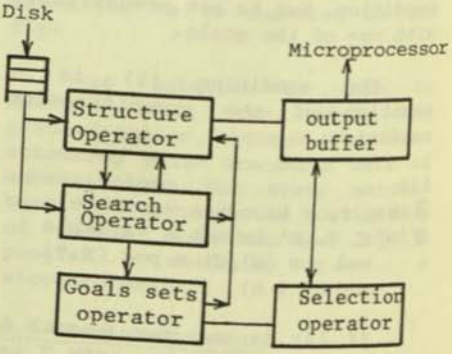


Figure 7.

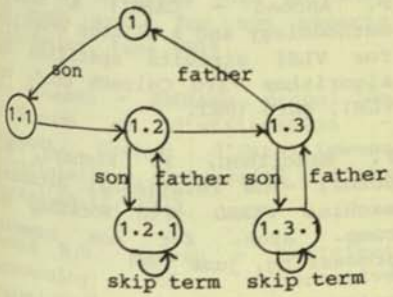
5.2. Description of the operators:

5.2.1. The structure operator:

It manages the input buffer, analyses the structure of the terms, and the types of data items. It uses a stack for the structure analysis, and a finite state automaton for the position encoding: the input characters are generated by the structure analysis and are the transitions to a son node, to a brother node or to the father node in the term. Every position which correspond to an element of a goal can be analysed and coded; the code is the address of a list of values in the search operator. When a variable is read, or when the data read coresponds to a variable in the goals, a substitution is transmitted in the output buffer: the substitution is composed of the data and its position. If we plan to recognize a maximum of 256 positions, the memory requirements of the automaton would be very small: about 10K bits.

Example:

goals: t (a , X , b)
t (c , d , X)



5.2.2. The search operator:

It receives from the structure operator the address of a list of values and the data read from the disk (5 bytes), and searches the data in the list. The index of the value will then be transmitted to the third operator. For a maximum of 256 values and 128 goals, a sequential search can be executed: the memory size will then be of 256 x 40 bits = 10 k bits. A fast static RAM can be used. With an access time of 40 ns, a list of 128 items can be searched in 5.12 microsecond, which is about the time necessary to read 5 bytes from a 10Mhz disk.

5.2.3. The set operator:

It manages a bit array where each column corresponds to a goal, and every row corresponds to the set of goals possible for an input value. The intersection of the successive rows is then executed for every address sent by the search operator. If the result is the empty set, a fail signal is sent to the other operators. Else, the result is decoded to get the list of goals which pre-unify with the input term. The size of the bit array will be of 256 x 128 bits (32 K bits).

5.2.4. The selection operator:

it removes from the output buffer the unusefull data, i.e. the substitutions corresponding to non preunifying goals.

5.6. Conclusion:

The total memory requirements for the first three operators is 52 Kbits (6.5 Kbytes), of which 10K must be static memories. It is then compatible with the VLSI implementation of the operators on a single chip with available technologies (the operators should be integrated with their memories). This memory size is relatively small, compared to other filtering techniques, such as finite state automata. The

memories can be loaded through DMA from the memory of the microprocessor: with a 2 Mbytes/s transfer rate, it will take about 3.25 ms. We can consider that the minimal interval between two disk accesses is 16.7 ms (one revolution), so that the compiling of the goals and the loading of the operators should take place during this interval.

Only the structure controler must follow the disk transfer rate, as it manages the input buffer. However, some operations will be executed faster, as they are only data transfers without processing, so that other operations can be executed a bit longer than the mean available time. The other operators do not normally process every data item and can execute at a slower rate.

The search operator could probably be implemented with slower memories than required for a truly "on the fly" operation. It is interesting to see that in such a case, sequential search can be a good solution, as it allows sufficient performances, greatly simplifies the operator, and reduces the memory size.

The set operator also does not need to be very fast. This will probably allow to execute the intersection in several steps. However, it will be interesting to check wether such a solution is more suitable than the use of a 128 bits AND operator.

Compared with other solutions for relationnal database systems, and particularly pure finite state automata, our solution appear more powerfull, as it allows full unification to be completed on the fly. Furthermore, its memory requirements are quite low: as an example, a tabulated automaton require a 256 X 256 array (64 K bytes), if the data is analyzed per byte, and for 256 transitions; hence the request can process at most 256 bytes (transitions). In our solution, with a tenth of the memory, we allow 128 X 5 bytes data items to be processed.

6. CONCLUSIONS:

The on the fly execution of the unification appears as an important feature of the OPALE project. A software simulation of the operators (including the goals compiler) is curently carrried out, the filter has been designed, and will be realized by this year.

An important topic is the parallel search strategy which is another significant part of our design efforts. We are planning for a complete software simulation of the machine, including search strategy and filtering. This will be executed on a multimicroprocessor machine. The VLSI implementation of the filter could also be investigated in the future.

REFERENCES:

F. Anceau - CAPRI: A design methodology and a silicon compiler for VLSI circuits specified by algorithms -3rd Caltech Conf. on VLSI, Marh 1983.

F. Bancilhon, P. Richard, M. Scholl -The relationnal database machine VERSO -6th Workshop on comp. arch. for non numeric processing, june 1981

U.S. Chakravarthy, J. Minker, D. Tran -Interfacing predicate logic languages and relationnal databases -1st int logic programming conference, Marseilles, september 1982

David j. De Witt - DIRECT - A mutiprocessor organization for supporting relational database management systems -IEEE Trans. on Computers, Vol C-28. No6, june 1979

J.S. Conery, D.F. Kibler - Parallel interpretation of logic programs -ACM Conf. on functionnal prog. lang. and Comp. arch. Portsmouth, October 1981.

H. Gallaire - Impacts of logic on data bases -VLDB 81

G. Gardarin - An introduction to SABRE multimicroprocessor data base machine -6th Workshop on comp. arch. for non numeric processing, june 1981

G. Berger Sabbatel, J.C. Ianeselli, Nguyen Gia Toan -A PROLOG data base machine -In "Data base machines", Springer Verlag, September 1983

J.A. Robinson - A machine oriented logic based on the resolution principle -JACM. Vol. 12, No 1, pp. 23-41, january 1965.

J. Rohmer - Associative filtering by automata: a key operator for data base machines -6th Workshop on comp. arch. for non numeric processing, june 1981

J. Roussel - PROLOG: manuel de référence et d'utilisation - Rapport, Groupe d'intelligence artificielle, Université d'Aix Marseille II, 1975

David H.D. Warren - Efficient processing of interactive relationnal database queries expressed in logic -VLDB 81

A PROLOG SYSTEM FOR THE VERIFICATION OF CONCURRENT PROCESSES AGAINST TEMPORAL LOGIC SPECIFICATIONS

P.G. Bosco, G.Giandonato, E.Giovannetti
Centro Studi E Laboratori Telecomunicazioni (C.S.E.L.T.)
Via Reiss Romoli 274
10148 Turin, Italy

ABSTRACT

A system, implemented in Prolog, for the verification of dynamic properties of concurrent processes is presented.

Descriptions of concurrent processes with asyncronous communication can be checked against dynamic behaviour specifications expressed by temporal logic formulas, under the hypothesis that the whole concurrent system can be modeled by a nondeterministic finite automaton.

We show the implementation for the basic components of the verifier: the model checkers for the chosen temporal logics, the symbolic simplifier, the dynamic semantics of the description language.

INTRODUCTION

This paper presents a prototype, written in Prolog, of a verifier which allows descriptions of concurrent systems, equivalent to nondeterministic finite automata, to be tested for the validity of dynamic properties.

Queries are formulated in the language of temporal logic, which permits the classical properties in the universe of concurrent programming, such as liveness, safety, deadlock absence, etc., to be expressed in a form that is both concise and sufficiently close to the intuitive concepts.

The language now used for the description of concurrent systems is SDL, the Specification and Description Language defined by C.C.I.T.T. (C.C.I.T.T. 1984), whose concurrency model is based on asynchronous communication through message queues.

We describe the implementations of the model checkers for linear-time and branching-time temporal logic; we also offer a survey of all the other components of the verification system, which shows how the structure of the system is general enough not to be dependent on the particular computational model imposed by the description language. It could therefore be used with a different language, or even in other fields of application, such as the verification of the properties of a hardware system.

1 THE COMPUTATIONAL MODEL

At this stage of our research the models we are interested in are the nondeterministic finite automata or the finite transition system (Clarke and Emerson 1981, Sifakis 1982), i.e. models consisting of a finite set of states on which a binary

accessibility relation r(S,S') is defined, expressing the reachability of the state S' from the state S trough one of the operations assumed as elementary in the actual system considered.

A concurrent system where each process can be modeled as a finite automaton (e.g. by only considering data types of finite cardinality) is in turn a finite automaton if communication is performed through bounded structures, like rendez-vous or bounded buffers.

In this case the global state of the system is the set of the local states of the processes plus the states of the memory elements possibly present in the communication structures.

The relation r among global states is obtained by modeling concurrency as the interleaving of the elementary operations of the processes.

2 TEMPORAL LOGICS

Different kinds of temporal logics have been used to describe concurrent system behaviour.

A major distinction is between branching-time and linear-time logics.
In the first case the structure on which the logic is interpreted is the state graph of the system; the truth value of a formula is defined for every state S in the graph and is a function - determined by the principal operator of the formula - of the truth values of its first-level subformulas in the states of the structure which are (immediately or by transitive closure) accessible from S.
In the second case the interpretation structure is originally a linear sequence of states, i.e. a path in the graph; therefore the truth value of a formula in a state S is relative to the chosen path starting with S, and depends on the truth values of the first-level subformulas in the states of the path. However, what one is often interested in is the validity of a formula in a state S (e.g. an initial state) relatively to all paths (starting with S), i.e. to all possible executions; so the linear formulas are usually intended as implicitly universally quantified over the paths.

We do not want, in this paper, to enter the debate linear vs. branching; without committing ourselves, for the time being, to one kind of logic, we have taken as references Clarke-Emerson's CTL (Clarke and Emerson 1981) and Manna-Pnueli's linear logic (Manna and Pnueli 1982,1983), respectively for the two points of view.

The primitive temporal connectives of linear-time logic are **x** (o), unary, and **u** (**until**), binary; **f** (◇) and **g** (□), unary, are derived connectives, definable in terms of the former. The primitive temporal operators of CTL are, using a variant of the original notation, **ex**, unary and **au**, **eu**, binary; **ax**, **af**, **ef**, **eg** are derived connectives.

We briefly recall their intuitive meanings:

x(F)
holds in a state S, on a path P starting with S, iff in the immediate successor of S on P F holds.

u(F1,F2)
holds in a state S, on a path P starting with S, iff there is on P a state S' where F2 holds, and in all states of P "preceding" S F1 holds.

f(F) = u(true,F)
holds in a state S, on a path P starting with S, iff there is on P a state S' where F holds.

g(F) = not(f(not(F)))
holds in a state S, on a path P starting with S, iff F holds in all states of P.

The branching-time operators are the universal and existential quantifications over paths of the corresponding linear operators, e.g.:

ag(F)
holds in a state S iff g(F) holds in S on all paths.

eg(F)
holds in S iff g(F) holds in S on some path.

.. etc.

3 MODEL CHECKERS

We have avoided, so far, tackling the complex problem of the construction of a system with deductive capabilities for these logics, which would suitably apply both to concurrent program verification and to synthesis from specifications.
Our present concern being mainly the verification of completely described systems, we have preferred the simpler approach based on model checkers, i.e. decision procedures which, given the state graph of the system and a formula, determine whether the graph structure is a model for the formula.

The "core" of the Prolog implementation, illustrated in the following paragraphs, is directly suggested by the fixed point characterization of the temporal operators, i.e.:

```
af(F) = F or ax(af(F))
ag(F) = F and ax(ag(F))
      . . .
```

These formulas are translated in Prolog by means of a **holds** predicate: so first order logic (Prolog) plays the role of a meta-language with respect to the object language: the temporal logic (Kowalski 1979).

3.1 BRANCHING-TIME LOGIC

The clauses for the operator ag are:

```
holds(S,ag(F)) :-                    (1)
  holds(S,[],ag(F)).

holds(S,H,ag(F)) :-
  in(S,H),!.

holds(S,H,ag(F)) :-
  holds(S,P),
  forall(r(S,S1),
         holds(S1,[S|H],ag(F))).
```

The variable **S** contains the state in which the temporal formula **ag(P)** is to be proved, where **F** is a generic subformula.

The only use of clause (1) is to initialize a variable employed during the computation to keep the path (history) from the initial state up to the current one; so a loop condition can be recognized as a double occurrence of the same state in the path, by means of the **in** predicate. As we are dealing with finite state automata, this situation eventually occurs on every non-ending path.

If the state components are constant values, **in** may be implemented as a **member** predicate, the path being represented by a list. In general, when symbolic values may be present in the state components, equalities have to be proved under more complex

constraints.

The **forall** predicate generates all possible successors of the state S, which are in turn to be tested for the validity of **ag(F)**.

```
forall(S,F) :- not forhelp(S,F).
forhelp(S,F) :- call(S),not(F).
```

The proof of **ag(F)** succeeds when F holds in every state and, for all paths, a loop node or a terminal node is reached. It fails as soon as a state in which F does not hold is encountered.

The clauses for the **af** operator (which is used, for example, to express "liveness" properties) are:

```
holds(S,af(F)) :-                    (2)
  holds(S,[],af(F)).

holds(S,H,af(F)) :-
  in(S,H),!,fail.

holds(S,H,af(F)) :-
  holds(S,F),!.

holds(S,H,af(F)) :-
  r(S,S1),!,
  forall(r(S,S2),
         holds(S2,[S|H],af(F)).
```

The proof succeeds when, for all paths, a state satisfying F is reached; otherwise, it fails if a loop where F is never satisfied is detected, or a terminal state is found.

The universal operators **ax**, **au** are handled in a similar way. The clauses for all the corresponding existential operators are obtained by the omission of the **forall** predicate. For example we have:

```
holds(S,ef(F)) :-
  holds(S,[],ef(F)).

holds(S,H,ef(F)) :-
  in(S,H),!,fail.

holds(S,H,ef(F)) :-
  holds(S,F),!.

holds(S,H,ef(F)) :-
  r(S,S1),holds(S1,[S|H],ef(F)).
```

The logical operators **and**, **or**, **not** can be decomposed as follows (if we limit ourselves to a propositional temporal logic):

```
holds(S,and(A,B)) :-
  holds(S,A),holds(S,B).

holds(S,or(A,B)) :-
  holds(S,A);holds(S,B).

holds(S,not(A)) :-
  not holds(S,A).
```

3.2 LINEAR-TIME TEMPORAL LOGIC

The model checker for linear-time temporal logic primarily deals with existentially quantified formulas like **Epath(F)**. This fits the existential mechanism of Prolog and avoids the explicit use of the **forall** metaconstruct. Universally quantified formulas like **Vpath(F)** are proved ab absurdo as not **Epath(not(F))**.

A possible "brute force" method for the linear logic could consist of: 1) generating a complete path; 2) applying on that path a set of clauses similar to those defining the branching-time existential operators.
For instance:

```
holds([S|Path],<>(P)) :-
  holds(S,P);
  holds(Path,<>(P)).
```

A major drawback of such a method is that complete paths are built even when the property could be proved on a subpath. We have then chosen to build the path incrementally as in the original branching-time clauses taking into account the fact that when the

proof of a formula splits into the proofs of the subformulas of a conjunction (occurring in the starting formula or generated by the recursive decomposition of temporal operators), all the and-components must be tested on the same path. So the subpath possibly instantiated by every subformula is imposed to the remaining subformulas. This behaviour is obtained as follows:

```
holds(S,|_|(F),Path,Pathnew) :-
 holds(S,[],|_|(F),Path,Pathnew).

holds(S,H,|_|(F),Path,Path) :-
 subpath([S|Path],H),!.

holds(S,H,|_|(F),
    Path,[S1|Path1]) :-
 holds(S,F,Path,Next),
 cbox(S,S1,H,F,Next,Path1).

cbox(S,S1,H,F,[],Path1) :-
 r(S,S1),
 holds(S1,[S|H],|_|(F),[],Path1).

cbox(S,S1,H,F,[S1|R],Path1) :-
 holds(S1,[S|H],|_|(F),R,Path1).
```

In order to prove **Epath** |_|(F) we start by proving **F** in the current state and since **F** could in turn be temporal it might instantiate a future subpath **Next**. Succeeding of **F** in **S** enables the recursive call of |_|(F) on a next state, which is the first in **Next** if Next is not empty, else it is generated by the accessibility relation r.

For the ◇ operator we have:

```
holds(S,<>(F),Path,Path1):-

holds(S,[],<>(F),Path,Path1).

holds(S,H,<>(F),Path,Path) :-
 subpath([H|Path],S),!,fail.

holds(S,H,<>(F),Path,Path1) :-
 holds(S,F,Path,Path1),!.

holds(S,H,<>(F),
      [S1|Path],[S1|Path1]) :-
 holds(S1,[S|H],<>(F),
       Path,Path1).

holds(S,H,<>(F),
      [],[S|Path1]) :-
 r(S,S1),
 holds(S1,[S|H],<>(F),
       [],Path1).
```

The logical operators **and**, **or** become now:

```
holds(S,and(A,B),Path,Pathnew) :-
 holds(S,A,Path,Path1),
 holds(S,B,Path1,Pathnew).

holds(S,or(A,B),Path,Pathnew) :-
 holds(S,A,Path,Pathnew);
 holds(S,B,Path,Pathnew);
```

Negation cannot be handled anymore by a clause like **holds(not(F),Path) :- not holds(F,Path)**, because the failure to find a path on which **F** holds is not the same as finding a path on which **not(F)** holds. The not operator has to be shifted, by the usual duality rules, inside the formula up to the elementary state predicates; only there negation can be proved by failure.

```
holds(S,not(<>(F)),Pth,Pth1):-
 !,holds(S,|_|(not(F)),Pth,Pth1).

holds(S,not(|_|(F)),Pth,Pth1) :-
 !,holds(S,<>(not(F)),
         Pth,Pth1).

holds(S,not(and(A,B)),Pth,Pth1) :-
 !,holds(S,or(not(A),not(B)),
         Pth,Pth1).

holds(S,not(or(A,B)),Pth,Pth1) :-
 !,holds(S,and(not(A),not(B)),
         Pth,Pth1).

holds(S,not(F),Pth,Pth1) :-
 not holds(S,F,Pth,Pth).
```

4 ATOMIC FORMULAS

If we restrict ourselves to a propositional temporal logic, the atomic formulas are propositional constants **p,q,...**; their truth values in every state **S**, may be defined by clauses of the form **p(S) :- ..., q(S) :- ..., ..** which are activated by:

```
holds(S,p) :- p(S).
holds(S,q) :- q(S).
    . . .
```

or, more concisely, by:

```
holds(S,F) :-                          (3)
  atomic(F),X=..[F,S],call(X).
```

The clause (3), where **F** is a metalinguistic variable ranging on the set of the above mentioned propositional constants, implements the transition from the stage where, through the intervention of a metalevel, written in Prolog, temporal formulas are evaluated, to the phase in which, all temporal constructs having been solved, we are merely in first order logic, directly handled by the Prolog interpreter.

For an extension to the predicative temporal logics, we must add the handling of quantified object-language variables.
With the constraint that the domain of the variables is the same in every state, an expression like **exist(x,F(x))**, where **F** is a generic temporal formula, holds in a state **S** iff there exists an individual **k** in the interpretation of temporal logic, i.e. an individual constant **k** of the object language, such that **F(k)** holds (Bowen and Kowalski 1982, Moore 1980). Instead of (3) we have then:

```
holds(exist(V,F)) :-
  objvar(V),
  substitute(V,X,F,FX),holds(FX).
```

V and **X** are two metalinguistic variables ranging respectively on object-language variables and object-language constants, which are both individual constants in the meta-language. The clause substitutes the variables **x, y, ...** of the object language with a Prolog variable **X** and, like (3), effects the shift from the metalevel to the basic Prolog level, the proper instantiation of the variable **X** being performed directly by the Prolog interpreter. Clauses for open formulas will be accordingly modified:

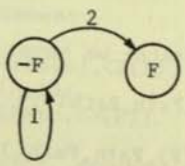
```
holds(S,p(X)) :- p(S,X).
holds(S,q(X,Y)) :- q(S,X,Y).
      . . .
```

or better:

```
holds(S,F) :-
  openatom(F),F=..[Op|Args],
  T=..[Op|[S|Args]],call(T).
```

5 FAIRNESS

In the model checker so far presented, the interpretations of some temporal expressions do not always correspond to their intuitive meanings (e.g. the "ideas" of a user interested in the verification of an actual transition system).
Let's look, for example, at the **af(F)** formula, expressing the eventuality of **F** and let's try to prove **af(F)** in the simple structure:



Since in the path **[-F,-F]** a loop is recognized which never verifies **F**, the proof of **af(F)** fails. It must be noted, however, that this answer would be right only if the nondeterministic choice between the

two transitions always fell on the first and never on the second even if both transitions are always enabled.
Such a behaviour should be avoided when dealing with actual nondeterministic systems, where transitions infinitely often enabled are known to be eventually executed, or more generally, infinitely often reachable predicates will eventually hold.
We are thus interested in a "fair" version of the af operator (in the sense of (Queille and Sifakis 1983)) which would validate the formula **af(F)**, in the previous example.

The clause (2) becomes:

```
holds(S,H,af(F)) :-
 in(S,H),left(H,S,Z),
 member(X,Z),r(X,Y),not in(Y,Z),
 holds(Y,H,ef(F)).

holds(S,H,af(F)) :-
 in(S,H),!,fail.
```

When entering a loop condition a check is performed, before failing, for the possibility of "exiting" the loop (selected by the **left** operation) by proving **ef(F)**, i.e. that F is reachable in at least one subpath starting with a state in the loop.

The same criterion is applied to eg(F), and to the $|\overline{\ }|$ operator in the linear logic, its model checker being based on the proof of existentially quantified formulas.

```
holds(S,H,|¯|(F),Path,Path) :-
 subpath([S|Path],H),left(H,S,Z),
 member(X,Z),r(X,Y),not in(Y,Z),
 holds(Y,H,<>(not(F)),
       [],Npath),
 !,fail.

holds(S,H,|¯|(F),Path,Path) :-
 subpath([S|Path],H),!.
```

These clauses for **Epath.**$|\overline{\ }|$**(F)** "rule out" a loop, in which F is always valid but the possibility to "exit" toward **not(F)** is permanently true.

6 DYNAMIC SEMANTICS OF THE CONCURRENT LANGUAGE

We show in this paragraph how to "program" the r relation in order to reflect the dynamic semantics of the concurrent language chosen for the applications, assuming that "interleaving" is a satisfactory model for such semantics.

The language we are focusing on is SDL (Specification and Description Language), defined by the C.C.I.T.T. as a standard in the telecommunication field. For the purposes of this paper it is sufficient to sketch the basic features of the language, which are common to other languages designed to describe concurrency.

A system can be composed by a fixed number of processes asyncronously communicating by means of queues, one for each process, containing the input messages for the process.
A basic execution cycle for a process is: 1) wait for an acceptable message in the queue; 2) execute a sequence of operations determined by the incoming message, until a next waiting point is reached; during this phase, called transition, send operations may be performed with the obvious meaning. During the "wait" phase a process can "save" messages in the queue, leaving them where they are, until it finds a message able to trigger a transition in the current waiting point. This message is the one actually consumed.
Messages on top of the queue, not declared in the waiting point (i.e. not "saved" and not firing a transition) are consumed without changing the process state.

As global state representation we use a triple **(Pcs,Bfs,Vrs)** where **Pcs** is a list of program counters each one of the form **p(Pname,Pc)**, where **Pname** is the process name and **Pc** its current "program counter"; **Bfs** is the list of the input queues, each represented by the term **b(Pname,[signal list])** and **Vrs** is the list of the local variable sets, each represented by **v(Pname,[...,(Vname,Value),...])**.

The main clause of the r relation for the above described concurrent system is:

```
r([Pcs,Bfs,Vrs],
  [Newpcs,Newbfs,Newvrs]) :-
  member(Mypc,Pcs),
  getname(Mypc,Pro),
  getvar(Pro,Vrs,Myvar),
  interp(Mypc,Mynewpc,Bfs,Newbfs,
         Myvar,Mynewvar),
  substvar(Pro,Vrs,
           Mynewvar,Newvrs),
  substpc(Pcs,Mypc,Mynewpc,Newpcs).
```

By means of the **member** predicate, a nondeterministic choice of the process to be "continued" is performed; **interp** is responsible for executing a transition leading from the current waiting point to the next one; **substvar** and **substpc** update the global state with the values deriving from the transition execution.

The **interp** predicate checks if there is an available message in the process queue. If so, the corresponding transition is executed, provided that the message was expected at the waiting point. Unawaited messages trigger an empty transition (4).

```
interp(p(Pro,wpoint(S)),
       p(Pro,Newpoint),
       Bfs,Newbfs,
       Myvar,Mynewvar) :-
  rcv(Pro,wpoint(S),Bfs,Bufl,Sid),
  interpl(Pro,S.Newpoint,Sid,
          Bufl,Newbfs,
          Myvar,Mynewvar).

interpl(Pro,S,Newpoint,Sid,
        Bfs,Newbfs,
        Myvar,Mynewvar)   :-
  declared(Pro,wpoint(S),
           input(Sid,N)),
  trans(Pro,Bfs,Newbfs,
        Myvar,Mynewvar,
        input(Sid,N),Newpoint).

interpl(Pro,S,wpoint(S),Sid,    (4)
        Bfs,Bfs,Myvar,Myvar) :-
  not declared(Pro,wpoint(S),
               input(Sid,N)).

trans(Pro,Bfs,Bfs,Vrs,Vrs,      (5)
      wpoint(S),wpoint(S)).

trans(Pro,Bfs,Newbfs,           (6)
      Myvar,Mynewvar,
      ifstmnt(Expr,N),Newpoint) :-
  eval(Expr,Value,Vrs),
  simplify(Value,Result),
  follows(Pro,ifstmnt(Expr,N),
          Result,Newop),
  exec(Newop,Bfs,Bfsl,Vrs,Vrsl),
  trans(Pro,Bfsl,Newbfs,
        Vrsl,Newvrs,
        Newop,Newpoint).

trans(Pro,Bfs,Newbfs,           (7)
      Vrs,Newvrs,
      Op,Newpoint) :-
  follows(Pro,Op,,Newop),
  exec(Newop,Bfs,Bfsl,Vrs,Vrsl),
  trans(Pro,Bfsl,Newbfs,
        Vrsl,Newvrs,
        Newop,Newpoint).
```

The last three clauses deal with the transition execution.
(5) is used to terminate a transition upon detecting the next waiting point.
(6) is an example of statement execution, namely the if-statement: the condition expression is first evaluated according to the current variable values and possibly simplified; then the branch selected by the result is followed.
(7) holds for the generic operation during a transition.

The send operation is performed by putting the signal Sid on top of the list (bottom of the queue) representing the queue of the addressed process.
rcv tries to receive a signal Sid from the process queue and eventually removes it.

```
snd(Sid,Pid,Bfs,Newbfs) :-
 subst(Bfs,b(Pid,Que),
     b(Pid,[Sid|Que],Newbfs).

rcv(Pro,Mypc,Bfs,Newbfs,Sid) :-
 member(b(Pro,Que),Bfs),
 extract(Sid,Que,Quel),
 not declared(Pro,Mypc,
          save(Sid,N)),!,
 subst(Bfs,b(Pro,Que),
     b(Pro,Quel),Newbfs).
```

The extract predicate:

```
extract(Sid,[X|Y],[X|R]) :-
 extract(Sid,Y,R).
extract(Sid,[Sid|Y],Y).
```

follows the FIFO discipline. The last element of the list Que (top of the queue) is extracted, and upon the next possible backtrackings (due to the fact that the selected signal is to be "saved" in the current waiting point) extract repeatedly picks the previous signals in the queue, until one acceptable in the current waiting point is found, or the queue is completely scanned.

7 THE SIMPLIFIER

As previously said, for loop detection we have to be able to recognize equalities between states; in addition, during the execution of an if-statement, we must obtain from the condition expression a result comparable with the branch labeling. When state components are constant values this equality could be solved by a simple syntactic comparison of terms. On the contrary, when symbolic values are present (corresponding, for instance, to initial, not specified values for variables), this method does not work.
For example, the boolean expression a, a and (a or b) are equivalent, but not syntactically equal, and the expression a -> (a or b) is equivalent to "true"; if the expression were the condition expression in an if-statement, we should reduce it to "true" in order to follow the "true" branch.
The general underlying problem is that of reducing expressions or "proving equalities" from axiomatizations of the involved data types (Huet and Oppen 1980).

For the time being, we have built a toy simplifier, based on rewriting rules, at present only dealing with booleans (according to the rules given in (Guttag et al. 1978)), but easily enrichable (once the good axiomatizations have been found) to treat nontrivial data-types.

Up to now, we have not deepened the problem of finding such "good axiomatizations" i.e. sets of rewriting rules having the Church-Rosser property. We have in mind, to this end, to use sophisticated versions of the Knuth-Bendix algorithm (possibly implemented in Prolog).
Here we show the "kernel" of the simplifier (a rewriting-rule activator, adapted from (Bergman and Deransart 1981)):

```
simplify(X,X) :- atomic(X).
simplify(X,Y) :- functor(X,F,A),
  bagof(Arg,analyze(X,A,Arg),S),
  Y1=..[F|S],!,normalize(Y1,Y).

analyze(X,N,Arg) :- upto(N,I),
  arg(I,X,Arg1),simplify(Arg1,Arg).

normalize(X,Y) :- rule(X,X1),!,
  simplify(X1,Y).
normalize(X,X).
```

```
upto(N,I) :- N > 0, N1 is N-1,
upto(N1,I).
upto(N,N) :- N > 0.
```

Rewriting rules are described as clauses like:

```
rule(leftside,rightside).
```

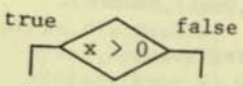
The simplifier works basically as follows: if the term to be reduced is atomic it is returned as a result; otherwise, the term being **f(x1,...,xn)**, the simplification reapplies to **f(y1,...,yn)**, with **x1,...,xn** previously reduced to **y1,...,yn.**
A sample of the rules for the boolean data type is:

```
rule(not(P),if(P,false,true)).
rule(and(P,Q),if(P,Q,false)).
rule(if(true,X,Y),X).
rule(if(false,X,Y),Y).
rule(if(C,true,false),C).
rule(if(if(P,Q,R),Lf,Ri),

if(P,if(Q,Lf,Ri),if(R,Lf,Ri))).
```

8 CONCLUSION AND FUTURE ENHANCEMENTS

The system in its present state cannot be used to handle descriptions of actual non-trivial concurrent systems, because it is too slow, due to the interpreter (CProlog on VAX) and to some design choices. It is however a useful prototype, which can be improved, without much effort, to a better efficiency and to a wider range of provable properties.
For instance, generating a complete state graph could speed up the verification of large sets of formulas. This can be obtained by running once for all the r procedure, and asserting its instantiations.
Efficiency could be further increased by storing for each state, during the recursive decomposition of temporal formulas, all valid subformulas. In this way, if the proof of a temporal operator has to "know" many times whether a subformula holds in a state, the validity test for such a subformula only takes place once, and leaves a "ground" clause in the Prolog data base, which is rapidly "matched" the next times.

We are now extending the system to handle conditional representations of states, deriving from the execution of if-statements where the exiting branch cannot be determined (because of the presence of symbolic values). For example in the following case:



if the variable **x** had the symbolic value **alfa** the next global state could be represented by the term **if(alfa > 0,S1,S2)**, where S1 and S2 are respectively the states deriving by the true and the false branches.

Efforts will be made to enlarge the set of data types handled by the simplifier and to find inductive rules allowing the proofs of particular properties to be carried out without generating the whole state graph.

The modularity of the system, i.e. the clean separation of the basic components, would hardly be achieved with a traditional language; moreover the backtracking mechanism of Prolog permits an easy implementation of nondeterministic concurrency models.

Prolog is particularly suitable for the development of prototypes of this kind of tools at a stage where theoretical issues are prevailing and the frequency of

conceptual rearrangement is high.

REFERENCES

Bergman, M.,Deransart, P. Abstract Data Types and Rewriting Systems: Application of the programming of algebraic data types in Prolog. CAAP81 Trees in Algebra and Programming 6th Colloqium. LNCS 112. March 81.

Bowen, K.A., Kowalski, R.A. Amalgamating object language and meta language in logic programming. Clark K.L., Tarnlund S.-A. eds., Logic Programming 153-172 Academic Press 1982.

C.C.I.T.T. 7/XI Report of the Paris meeting on the formal definition of SDL 1984.

Clarke, E.M., Emerson, E. Design and Synthesis of Syncronization Skeletons using Branching-time Temporal Logic. Proc. of the Workshop on Logic of Programs. LNCS 131 1981.

Emerson, E., Halpern, J.Y. "Sometimes and not never" revisited: on branching versus linear time. Proc. of 10th POPL 127-140 1983.

Fujita, M., Tanaka, H., Moto-oka, T. Verification with Prolog and Temporal Logic. 6th International Symposium on Computer Hardware Description Languages, Barbacci, Uehara eds., North-Holland 1983.

Guttag, J.V., Horowitz, E., Musser, D.R. Abstract Data Types and Software Validation. CACM 21, No. 12, 1978.

Huet, G., Oppen, D. Equations and Rewrite Rules: a Survey. Book R. ed., Languages: Perspectives and Open Problems. Academic Press 1980.

Kowalski, R.A. Logic for Problem Solving. North-Holland 1979.

Manna, Z., Pnueli, A. Verification of Concurrent Programs: The Temporal Framework. Boyer R.S., Moore J.S. eds., The Correctness Problem in Computer Science. 215-273, Academic Press 1982.

Manna, Z., Pnueli, A. Proving Precedence Properties: The Temporal Way. Stanford Report STAN-CS-83-964, 1983.

Moore, R. Reasoning about Knowledge and Action. Technical Note 191, SRI International, 1980.

Queille, J.P. and Sifakis, J. Fairness and Related Properties in Transition Systems. A Temporal Logic to Deal with Fairness. Acta Informatica 19, 195-220, 1983.

Sifakis, J. A Unified Approach for Studying the Properties of Transition Systems. TCS 18, 227-258, 1982.

LOGICAL LEVELS OF PROBLEM SOLVING

Leon Sterling
Department of Applied Mathematics
Weizmann Institute of Science
Rehovot 76100, Israel

ABSTRACT

This paper demonstrates how clear, efficient problem solving programs can be written within logic programming. The key point is the consideration of levels involved, both in the problem solving itself and in the underlying logic. Three levels of knowledge necessary for intelligent problem solving are identified - a level of domain knowledge, a level of methods and strategies, and a planning level. The approach introduced here relates these levels to the distinction between object and meta languages. Two classes of programs are presented. Firstly, single level problem solvers are introduced. These are at the methods level and constitute a meta language of the problem domain. Finally flexible multi level problem solvers are outlined which can be built as extensions of the single level programs.

1 INTRODUCTION

There are many different pieces of knowledge needed to build powerful problem solvers. Knowledge about the domain, knowledge about the available problem solving methods and strategies, knowledge about forming plans from the methods. This paper claims that distinct levels should exist for the different types of knowledge, and shows how to incorporate this differentiation of levels into clear, efficient problem solving programs.

Three levels are introduced – a *domain level*, a *methods level* and a *planning level*. No formal definition will be given of these and boundaries between them are somewhat fuzzy. However the three levels have a hierarchic relationship, where the methods level sits between the planning level and the domain level. A similar decomposition into levels has been given by Stefik (Stefik 1981) in his program for planning experiments in molecular genetics. He describes three spaces, a strategy space, a design space and laboratory space, which correspond respectively to the planning level, methods level and domain level introduced here.

In order to discuss problem solving one needs a language. Predicate logic is chosen here as the language for representing both the task of problem-solving and the more abstract entities such as strategies and plans. The case for using logic as the representational language for problem-solving has been presented strongly in other places, e.g. (Hayes 1977) and (Moore 1982). Their principal point is that all reasoning performed should be explicitly represented and not hidden in fancy data or control structures.

In this paper the further restriction to Horn clause logic is made.

Logic programming has two aspects making it ideal for problem solving – a clear semantics as advocated in the previous paragraph, plus a practical language, Prolog, for implementing the problem solvers. All examples in this paper of problem solving programs will be given as Prolog code. Background on the use of logic for problem solving can be found in (Kowalski 1979), whilst for programming in Prolog the reader is referred to (Clocksin and Mellish 1981).

Choosing logic as the representational language gives rise to further levels – those arising from the notions of object and meta language. Trying to use the meta language in problem solving programs is not new. Using the power of meta-level reasoning has always been a seductive idea waiting to be exploited. Several researchers have discussed how meta concepts could be incorporated into intelligent programs. Weyhrauch (Weyhrauch 1980) gives a formal treatment and describes a system where interaction between the levels, termed reflection, happens. Bowen and Kowalski (Bowen and Kowalski 1982) discuss reflection in the context of logic programming, considering systems which amalgamate object and meta languages. Bundy and coworkers have discussed 'meta-level inference' for problem-solving, describing programs solving symbolic equations (Bundy and Welham 1981), and mechanics problems stated in natural language (Bundy et al. 1979) which are based on distinguishing between object and meta theories. Davis (Davis 1980) and Stefik (Stefik 1981) discuss including meta concepts in expert systems.

This paper takes a new approach, relating the object/meta level distinction to the levels of problem solving. A key idea is the association of the methods level with a meta language of the domain level. More traditionally, the meta language of the problem solving domain is associated with control, e.g. (Davis 1980). Even Bundy and Welham (Bundy and Welham 1981) who essentially axiomatize a methods level for equation solving present their work in terms of controlling search of an object level space. Clarifying the problem solving levels here puts the work described above in perspective. And, as will be demonstrated, the approach advocated here leads to powerful, practical problem solvers.

A brief diversion is appropriate to relate this paper to the development of expert systems. I regard expert systems as a special case of problem-solving programs. Hence lessons from research in expert systems are relevant for writing problem solving problems in a logic programming framework and vice versa. The innovation provided by research in expert systems was essentially due to the sort of problems tackled and the approach of achieving expert performance by making programs knowledge intensive. Expert systems have been successful in domains where the inference necessary is relatively superficial, but the domain knowledge is of key importance. Earlier research on problem-solving on the other hand, for example GPS (Ernst and Newell 1969), concentrated on identifying, developing and using powerful, general-purpose inference mechanisms. These are artificial distinctions particularly as expert systems develop. It has been argued eloquently by Davis in (Davis 1980) that expert systems will need to become more sophisticated in their reasoning.

The layout of the paper is as follows. The next section introduces the

object and meta levels in an informal manner. These are illustrated in the following section with a simple example, the plan-formation problem for the blocks world discussed by Kowalski in (Kowalski 1979) and (Kowalski 1981). The following two sections discuss how problem solving programs should be written in logic programming as influenced by these notions of levels. Section 4 introduces single level problem solvers. These are practical, efficient programs written at the methods level, rather than at the domain level, in contrast to most expert systems. A methodology for writing such single level programs is given in (Sterling 1984). Section 5 shows how more flexible multi level problem solving programs can be built in logic programming. Finally brief conclusions are given.

2 OBJECT AND META LEVELS

The aim of this section is to clarify the meaning of the terms 'object' and 'meta'. A ***language*** consists of a ***theory***, that is a set of axioms, and a ***proof procedure***. The object-level of the problem-solving domain is a typical language. A proof procedure consists of a strategy for enumerating the inferences that can be made within the theory, and a computation rule for resolving the non-deterministic choices of the strategy. An interpreter is needed to execute the proof procedure. A pure logic program constitutes a language – the Horn clauses (given their declarative reading) being the theory and SLD resolution (augmented by a suitable computation rule) the proof procedure. Prolog interpreters execute this proof procedure using depth first search as the computation rule.

Two languages L1 and L2 are in an ***object-meta*** relationship if there is a (partial) axiomatization of the theory and proof procedure of language L1 in the language L2. In this case the terms object theory, object language, meta theory and meta language have their obvious interpretation. Informally the meta language describes the relationships that hold in the object language. Examples will be given in the next section.

Introducing a meta theory for an object language gives an additional way of solving an object level problem. Not only can the object level proof procedure be executed directly, but it can be ***simulated*** via its representation in the meta language. More generally, a proof can contain inferences in both the object theory and meta theory. Crossing between the object and meta levels has been termed ***reflection*** and discussed formally in (Weyhrauch 1980). It is shown that solving problems by direct execution in the object language is equivalent to simulation in the meta language. There is a detailed discussion in (Kowalski 1979) of object and meta languages, simulation and reflection, in the context of logic programming.

The remainder of the paper discusses how to relate these logical entities to the problem solving levels. Briefly, the most powerful effect occurs when the methods level is regarded as a meta language for the domain level. It is also possible to regard the planning level as a meta language of the methods level, but that relationship remains to be explored.

3 PROBLEM SOLVING IN THE BLOCKS WORLD

This section considers an example to illustrate the various levels introduced so far. The example used is

a variant of the ***plan-formation*** problem discussed in (Kowalski 1979) and (Kowalski 1981). The problem is to form a plan in the blocks world, that is to specify a sequence of actions for restacking blocks to achieve a particular configuration. An approach concentrating on the planning level can be found in (Warren 1974) where a general planning program is given in Prolog capable of solving blocks world problems.

Figure 3.1 gives the initial state and desired final state of a blocks world problem. The actions allowed are moving a block from the top of a block to a place, and moving a block from one block to another. For the action to succeed the top of the moved block must be clear, and also the place or block to which it is being moved.

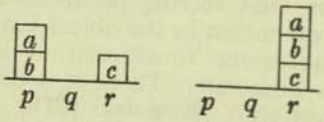


Figure 3.1: Initial and final states of a blocks world problem

Kowalski gives, in his own words, both a one-level and two-level formulation. The one-level formulation in (Kowalski 1979) is essentially a domain level specification. In (Kowalski 1981) however, he "employs a two-level representation using the object-level to describe the individual states of the database and using the metalevel to describe the relationship between one state of the database and a successor state."

Figure 3.2 gives a program for solving the plan formation problem. We discuss it in the light of the definitions of the previous section, comparing it with Kowalski's versions. No attempt has been made to improve the power of Kowalski's formulation. The only changes that have been made are to clarify the manifestation of levels as will be discussed below. Edinburgh Prolog conventions are used throughout, see for example (Clocksin and Mellish 1981).

```
plan_form(Plan) :-
    initial_state(S1),
    state_trans(S1,S2,[],Plan),
    final_state(S2).

state_trans(S,S,Actions,Actions).
state_trans(S1,S2,Ac,Actions) :-
    update(S1,A,Ac,S),
    state_trans(S,S2,[A|Ac],Actions).

update(S1,A,Ac,S2) :-
    action(A,S1), legal(A,S1),
    not member(A,Ac),
    transform(A,S1,S2).

action(to_block(X,Y,Z),S) :-
    on(X,Y,S), block(Z).
action(to_place(X,Y,Z),S) :-
    on(X,Y,S), place(Z).

legal(to_block(X,Y,Z),S) :-
    clear(X,S), clear(Z,S), X \== Z.
legal(to_place(X,Y,Z),S) :-
    clear(X,S), clear(Z,S), X \== Z.

initial_state([on(a,b),on(b,p),on(c,r),
    clear(a),clear(q), clear(c)]).
final_state(S) :-
    member(on(a,b),S),
    member(on(b,c),S),
    member(on(c,r),S).

clear(X,State) :-
    member(clear(X),State).
on(X,Y,State) :-
    member(on(X,Y),State).

block(a).   block(b).   block(c).
place(p).   place(q).   place(r).
```

Figure 3.2 : Program for the plan-formation problem

Key procedures are state_trans and update. The predicate state_trans(S1,S2,Ac,Plan) is true if there is a plan of actions, Plan, transforming state S1 into state S2. Note that S1 and S2 name states. The third argument, Ac, is an accumulator of the actions performed so far, necessary to avoid looping through previous states. A more powerful problem solver would keep a list of former states rather than former actions. This introduces the problem of determining when two states are identical and is beyond the scope of Kowalski's program. The predicate update(State,Action,Ac,NewState) is true if State names a state, Action an action, Ac the actions performed so far and NewState names the state obtained by applying Action to State. Attempting to satisfy the update goal simulates the performance of the action in the blocks world.

The names chosen here to represent states are very descriptive – just a list of the facts which are true. For example the 'name' of the initial state is [on(a,b), on(b,p), on(c,r), clear(a), clear(q), clear(c)]. Such names have an advantage that they allow easy testing whether facts are true in the states being named. For example to know whether a particular block, X say, is clear in state S, one tests whether the fact clear(X) is a member of the list of the name of state S. The predicates clear and on have been thus defined in figure 3.2.

This is slightly different from Kowalski's approach, who introduces a predicate demonstrate(X,Y) (demo for short) which for the current problem is true if Y is true in state X. For example, to test whether a state S is a final state would be written as

```
final_state(S) :-
    demo(S,G1), demo(S,G2).
```

where G1 and G2 name the terms you want to be true, namely on(a,b) and on(b,c). Kowalski also uses the demo predicate to make explicit the use of reflection. Using demo avoids the need of defining separate test predicates for each predicate such as clear and on, but is likely to yield inefficient programs if all operations must go via the demo predicate. Such considerations will become less important if work on automatic program transformation advances.

In the program of figure 3.2 changing the state of the world is done by the procedure transform(Action,S1,S2). S1 is the list of facts used to name state S1, while S2 is similarly the list of facts naming S2. There are different ways of writing it, but all essentially just remove the facts from the list S1 that are no longer true and add the lists that have become true to obtain the list S2. Kowalski similarly updates the global database via add and delete predicates.

How do the different levels appear in this example? The manifestation of logical levels has already been indicated. The predicates update and state_trans are inherently part of a meta language of the blocks world rather than of an object language. Further the problem of forming a plan has been solved by simulation in the meta language rather than direct execution in the object language.

The distinction between simulation and direct execution may not be obvious here due to the complete axiomatization of the object language proof procedure, moving blocks to change the world, in the meta language. The program in figure 3.2 axiomatizes the movement of the blocks with the predicate update which calls the predicate action to

find all the possible actions. Here they are just the simple domain level operators. In general only a partial axiomatization might be given. The examples of the next section have an incomplete axiomatization of the object language proof procedure in the meta language. This leads to efficient programs for solving equations and proving theorems.

Where do the problem solving levels fit in? This is not a rich example in terms of knowledge needed to solve the problem. The methods level consists of only two simple methods, moving a block to a block and moving a block to a place, which are a direct translation of object level operators. (Kowalski only has one method, the extra one here is only for illustrative purposes). A methods level more generally would consist of more interesting strategies not necessarily directly obtained from object level operators. For the blocks world such methods are reversing the order of a stack of two blocks, or building a tower. These methods would be axiomatized in the same way as to_block and to_place.

Structure or knowledge related to planning is also minimal in the program. Further it is present implicitly rather than explicitly. Examples of such planning knowledge are structure of methods and preferred order of methods. The methods can be specified for example in terms of add and delete lists, namely the list of facts made true by the method, and the list of facts no longer true. Here they are built into the transform predicate. The to_block action is preferred to the to_place action indicated by its appearing first in the list of actions. So the planning level doesn't explicitly exist here. It is shown in section 5 how such implicit levels can be made explicit.

4 SINGLE LEVEL PROBLEM SOLVERS

In this section we consider how these notions of levels get translated into problem solvers. The naive view, adopted by the early expert systems, was to have only one level, a domain level full of knowledge, and a simple proof procedure to find consequences of that knowledge. Concepts of object and meta language were not considered, and the problem solving domains were such that methods and plans were not particularly necessary. Davis (Davis 1980) claims that such an approach is fundamentally limited.

A more promising approach, still only using one level, is to think in terms of methods rather than domain knowledge. The program of the previous section was a toy example of such a program. When a reasonably comprehensive set of methods can be found for a problem solving domain, a powerful single level problem solver can be built. In terms of the three problem solving levels introduced in this paper, only the middle one, the level of methods and strategies, is actually present. However, by incorporating the concept of object and meta languages one can understand where the other levels fit in. Axiomatizing a meta theory rather than an object theory to aid in problem solving has been described in (Bundy and Welham 1981).

What happens to the other levels in such single level problem solvers? The domain level becomes axiomatized in the methods level, that is the methods level constitutes a meta language of the problem solving domain. All problem solving then occurs via simulation in this meta language. The power of the problem solver then depends on this axiomatization. In general there is

a trade-off between efficiency gained from only partially axiomatizing the domain level and completeness of the final problem solver. In principle one can simulate the complete object level proof procedure in the meta language but this leads to inefficient programs. Axiomatizing complex strategies in terms of simple domain actions leads to efficient programs.

The planning level is treated differently. It becomes 'programmed' into the methods level using knowledge of the behaviour of the interpreter for the methods level. For writing problem solvers in Prolog, this means using the order of clauses in a procedure, and the order of literals in the body of a clause to convey the planning information. For sufficiently simple domains, this planning information can be expressed cleanly. In (Sterling 1984) a methodology is described for building these single level problem solvers.

In order to gauge the appropriateness of this approach, let us consider some examples. A powerful single level problem solver has been built for solving symbolic equations. The program, PRESS, has been written in Prolog and described in (Bundy and Welham 1981) and (Sterling et al. 1982). PRESS regards the domain level as manipulating algebraic formulae according to rules of algebra. Methods are applications of rewrite rules to produce a particular effect, for example isolating the occurrence of a variable on the left hand side of an equation. Plans are sequences of methods. Arguments establishing the advantages of using a meta language for the particular case of equation solving have been made in other places. The benefit of using simulation at the meta level to solve symbolic equations has been argued in (Bundy and Welham 1981), while how the equations are then solved by simulation is explained in (Bundy and Sterling 1981). The equation solving abilities themselves are described in (Sterling et al. 1982). It should be noted however that hindsight allows a new perspective on what has been done, and so the description of the approach to equation solving suggested by PRESS is expressed differently here.

Another example is the task of theorem proving. Traditionally theorem provers, especially those based on resolution, were domain based. Inference rules and axioms for a theory are given and theorems are proved by applying the inference rules to the axioms. It is hard to express strategies in such theorem provers and the few impressive theorems they have proved have been accomplished by brute force - see for example (Overbeek, McCharen and Wos 1976).

The approach suggested here is to add a methods level and axiomatize the theorem proving steps within this methods level. What are the methods in theorem proving? An example is induction. Figure 4.1 gives a sample of program code for implementing this method. It illustrates the power of single level programs, but also expresses the planning information. More details about the full program, and the particular induction proof plan it implements, can be found in (Sterling and Bundy 1982) and (Sterling 1982).

(i)

```
prove_by_induction(Thm) :-
    struct_induction(Thm,Scheme),
    prove_base_case(Thm,Scheme),
    prove_step_case(Thm,Scheme).
```

```
(ii)
prove_step_case(Thm,Sc) :-
    step_version(Thm,Sc,StepV,
      ProgHyp),
    neg_and_skol(StepV,G,Ass ),
    ind_hyp(Thm,Sc,IndHyp),
    prove(G,Sc,Ass,ProgHyp,
      IndHyp).

(iii)
prove(G,Sc,Ass,ProgHyp,IndHyp) :-
    unfold_hyp(ProgHyp,Perf,Rec),
    fold_goal(G,GPerf,GRec),
    apply_ind_hyp(GRec,IndHyp,G1),
    establish_hyp(G1,Sc,Rec,G2),
    add_goal(G2,GPerf,G3),
    establish_step(G3,Ass,Sc,
      Perf,Rec).
```

Figure 4.1 : A fragment of a program to prove theorems by induction

How are the levels manifest in this code? The planning level is incorporated in two ways. Firstly by the order of literals in the bodies of clauses. Clause (i) in figure 4.1 tells how to prove a theorem by induction, namely you must find a suitable induction scheme, prove the base case, and prove the step case. Before proving the base and step cases, it is necessary to find the appropriate induction scheme. This information is built into the order of the literals in the body of clause (i). Secondly, clauses are ordered in their preferred order of use. This is not illustrated in the code above, but will be discussed in the next section in the context of the relationship between single and multi level programs.

The actual theorem proving steps or domain level transformations are axiomatized in tactics which express the relationship between various logical formulae. An example of a tactic is 'fold', introduced by Burstall and Darlington for programs expressed as first order recursion equations (Burstall and Darlington 1977). Bundy adapted it for simply recursive logic programs – the adaptation is described in (Sterling and Bundy 1982). Particular instances have the form fold(Clause,Recurse,New), where New is the result of folding Clause with respect to its recursive definition, named by Recurse. Attempting to satisfy a fold goal will result in the fold transformation taking place by simulation. How that is implemented is irrelevant to the problem solver, all that is needed to know is that a fold step has been done. The predicate, fold_goal, appearing in clause (iii) in figure 4.1 invokes the fold tactic for example.

The structure of the code of figure 4.1 is a result of polishing the methods of theorem proving. There has been an organization step where the plan has been clarified and refined to such an extent that it is readily expressible in a one level program. It should be noted that such polishing proceeds naturally by top-down program development. The top-down nature of the development is due to thinking at the methods level rather than the domain level.

Sometimes a single level problem solver suffices for the problem solving task. This occurs if a powerful set of methods can be found for the domain. In this case, there seems to be little point in adding extra levels of complexity, such as planning, if they are not needed in solving the problem. For example, it is shown how to add the framework for a planning level for equation solving in the next section. If it doesn't improve the equation solving ability of the program there is no need to incorporate it.

What conclusion can the builder

of an expert system draw from this? How can he determine a priori whether his domain will be sufficiently simple that a one level program will suffice? There is no obvious answer, but nothing is lost by writing as if the domain were simple. My experience leads to the conjecture that the clarifications of difficulties in the problem solving domain that will be raised by trying to write a one level program will be directly useful when writing a multi level program. Writing programs in this way is very much in the style of structural development (Sandewall 1978).

5 MULTI LEVEL PROBLEM SOLVING

Single level problem solvers are inadequate in general. The uniform proof procedure needed to execute them is usually too inflexible. This inflexibility was commented on in the early days of automatic theorem proving by many, for example Hayes (Hayes 1973). He believed in controlling logic with logic, a view agreed with here. The problem is how to implement it.

One approach is to add an extra control level. The earliest proponent for this approach in expert systems is Davis (Davis 1980). He argues for introducing 'metarules' as a way of adding control. In terms of the levels discussed here, it means adding a meta language to the domain level within which one expresses control of the domain level proof procedure. Figure 5.1 gives a metarule used by Davis in (Davis 1980) and suggests its translation in logic programming. The translation is not exact due to the different contexts. For example, considering rules with uncertainties, important in the expert systems studied by Davis, is not part of the general problem solving style in logic programming.If

[1] the age of the client is greater than 60,

[2] there are rules which mention high risk,

[3] there are rules which mention low risk,

then it is likely (.8) that the former should be used after the latter.

```
prefer(Rule1,Rule2,Context) :-
    low_risk(Rule1),
    high_risk(Rule2),
client_age(Age,Context), Age > 60.
```

Figure 5.1 : A meta rule of Davis expressed in logic

This approach of equating the meta level with a general control level has been adopted into logic programming by Gallaire and Lasserre (Gallaire and Lasserre 1982). They have defined a set of control primitives and propose adding metarules to programs to improve performance in the way advocated by Davis. Writing an appropriate interpreter is another way of adding a control level. In (Pereira 1982) a methodology is given for writing interpreters in logic for controlling logic programs.

Using the meta language as a control language, however, means piecing together a program from general-purpose tools and techniques rather than building an integrated, domain specific program. The message of the expert system experience is that using domain knowledge is important. It seems reasonable that specific knowledge will be vital for writing the planning and methods levels as well as the domain level. Information such as the order in which to apply methods should be incorporated into the problem solving predicates themselves rather than in gen-

eral purpose primitives. When developing programs I find it more natural to think in terms of domain concepts rather than somewhat artificial control primitives.

A better alternative to adding a control level is embedding the single level problem solver in a multi level version. The different forms of knowledge (about plans, methods and the domain) can be reasoned about in distinct ways. In principle each knowledge level added requires its own interpreter. The problem is incorporating the different interpreters efficiently. We now discuss how this can be achieved, principally considering how the equation solving program, PRESS, mentioned in the previous section, could form the basis of a multi level program.

Let us consider the addition of a planning level to the equation solving program. The top-level goal of PRESS is a procedure solve(Equation,X,Solution). This procedure is essentially the interpreter for the equation solving methods, and is executed by the Prolog interpreter. To begin at the planning level we similarly need a top level procedure for the planning level which can again be executed by Prolog. Some simple code for the task is given in figure 5.2.

```
plan_solve(Eqn,X,Solution) :-
    applicable_plans(Eqn,X,Plans),
    choose_plan(Eqn,Plans,Plan),
    solve(Eqn,X,Solution,Plan).
```

Figure 5.2 : Interpreter for a planning level for equation solving

The predicate plan_solve builds in the possibility for sophisticated planning. The first predicate, applicable_plans, would find the suitable plans for the equation. These might be taken from a library of available plans, or generated on the basis of the features of the particular equation. Specific planning information about equations would be used when appropriate. The next predicate, choose_plans, would filter the plans arising from the previous stage, and select one. Various heuristics could be encoded as to selection of plans. The predicate, solve/4, is then the interface to the methods level where the equation is solved. It replaces the predicate solve/3 as interpreter of the methods level.

How a plan would be represented and how it would be used by a methods level would vary from domain to domain. For equation solving a simple representation of plans is sufficient to achieve the performance of PRESS. I added a toy planning level to PRESS where plans were represented by the structure plan(Name, PreConditions, MethodSteps). The preconditions were used in the choice of appropriate plan. The predicate solve was written as

```
solve(Equation,X,Solution,Plan) :-
    plan(Plan,_,MethodSteps),
    call(MethodSteps).
```

In fact the appropriate method steps were taken directly from the solution procedure of PRESS. This primitive planning level did not improve the equation solving behaviour of PRESS – to the contrary it slowed the program.

The knowledge oriented multi level approach advocated here is not universally accepted. In their paper (Gallaire and Lasserre 1982) on meta level control, Gallaire and Lasserre give a brief survey of different approaches. Our view is closest to what they call knowledge structuring. They disregard it after saying

the following. "Instead of talking in terms of interpreter behaviour, they talk in terms of levels of perception of the world (e.g. objects, assemblies, equations, heuristics,...). As no general agreement has yet been reached on a world structuring language, they are led to build their own language and interpreter."

What they imply is a flaw I claim as a feature. The difficulty in building problem solvers is representing the relevant knowledge. In logic programming that means axiomatizing the problem domain. Coming up with an appropriate axiomatization is essentially building a language. Solving problems from different domains usually requires different views of the world, and hence different languages. It is also true that in principle each domain needs its own interpreter. But building an interpreter in Prolog is not an expensive overhead. Both the interpreters, for the planning and methods level described above, are Prolog programs where the difficulty is in understanding the domain, not in specifying the control.

Consider another example taken from (Silver 83) where a program, LP, is described capable of learning to solve equations from worked solutions. Given a worked example, LP builds a schema for solving the equation, which consists of a list of methods to apply. To solve a new equation the predicate schema_solve is called, a simplified version of which appears in figure 5.3.

```
schema_solve(Eqn,[M|Ms],X,Ans) :-
    apply_method(M,Eqn,X,New),
    schema_solve(New,Ms,X,Ans).
schema_solve(Eqn,_,X,Ans) :-
    solve(Eqn,X,Ans).
```

Figure 5.3: Applying a learned schema to a new equation

The predicate schema_solve is at the planning level of equation solving. It decides to try to apply a schema first, and if that is unsuccessful calls the usual solve procedure. In order to apply a schema, one must apply the methods contained in the schema. How to apply methods, the predicate apply_method, is again at the planning level, where the relevant information can be appropriately expressed. Thus LP can be regarded as a multi level extension of PRESS.

A similar evolution is possible from the theorem proving program described in figure 4.1. A program is being developed (Wallen 1983) which expresses the fine detail of an induction proof plan. The initial program had insufficient expressive power to control the theorem proving process, and thus the single level program is being expanded into a multi level one.

6 CONCLUSIONS

We have discussed various notions of levels of problem solving – arising both from the problem solving task itself and the logic used to implement it. Powerful single level problem solvers are presented where the relationship to the other levels is made clear. Such programs are at the methods level, and constitute a meta theory for the domain. Intelligent, flexible problem solving requires inferences to be made at several levels. It is shown how the single level problem solvers can be embedded in multi level problem solvers, where implicit knowledge is made explicit.

ACKNOWLEDGMENTS

I would like to thank Ehud Shapiro for most helpful suggestions on earlier versions of this paper. The referees were also responsible for im-

provements. The author is currently supported by a Dov Biegun Postdoctoral Fellowship.

REFERENCES

Bowen, K. and Kowalski, R. Amalgamating language and meta-language. Logic Programming, Academic Press, 1982.

Bundy, A. and Sterling, L. Meta-level inference in Algebra. Research Paper 164, Department of Artificial Intelligence, University of Edinburgh, 1981.

Bundy, A. and Welham, B. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. Artificial Intelligence 16, 189-212, 1981.

Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R. and Palmer, M. Solving mechanics problems using meta-level inference. Proceedings of IJCAI-6, pp. 1017-1027, Tokyo, 1979.

Burstall, R. and Darlington, J. A transformation system for developing recursive programs. J. ACM 24, 44-67, 1977.

Clocksin, W.F. and Mellish, C.S. Programming in Prolog. Springer-Verlag, 1981.

Davis, R. Meta-rules: Reasoning about control. Artificial Intelligence 15, 179-182, 1980

Ernst, G.W. and Newell, A. GPS: A Case Study in Generality and Problem Solving. Academic Press, 1969.

Gallaire, H., and Lasserre, C. Metalevel control for logic programs. Logic Programming, pp. 173-185, Academic Press, 1982.

Hayes, P. Computation and deduction. Proceedings MFCS Symposium, Czech. Academy of Sciences, 1973.

Hayes, P. In defence of logic. Proceedings IJCAI-5, pp. 559-565, MIT, 1977.

Kowalski, R. Logic for Problem Solving. North-Holland, 1979.

Kowalski, R. Prolog as a logic programming language. Proceedings AICA Congress, Pisa, 1981.

Moore, R. The role of logic in representation and commonsense reasoning. Proceedings AAAI-82, pp. 428-433, 1982.

Overbeek, R., McCharen, E. and Wos, L. Complexity and related enhancements for automated theorem-proving programs. Comp. and Maths. with Appl. 2, 1-16,1976

Pereira, L.M. Logic control with logic. Proceedings First International Logic Programming Conference, pp. 9-18, Marseille, 1982.

Sandewall, E. Programming in an interactive environment: the LISP experience. Computing Surveys 10, 1978.

Silver, B. Learning equation solving methods from examples. Proceedings IJCAI-8, pp. 429-431, Karlsruhe, 1983.

Stefik, M. Planning and meta-planning (MOLGEN: Part 2). Artificial Intelligence 16, 141-170, 1981

Sterling, L. IMPRESS – Meta-level concepts in theorem proving. Working Paper 119, Deparment of Artificial Intelligence, University of Edinburgh, 1982.

Sterling, L. Implementing problem-solving strategies using the meta-level. Proceedings 4th Jerusalem Conference on Information Technology, May 1983.

Sterling, L., Bundy, A., Byrd, L., O'Keefe, R. and Silver, B. Solving symbolic equations with PRESS. Computer Algebra, LNCS 144, pp. 109-116, Springer-Verlag, 1982.

Sterling, L. and Bundy, A. Meta-level inference and program verification. Proceedings Sixth Conference on Automated Deduction. LNCS 138, pp. 144-150, Springer-Verlag, 1982.

Wallen, L. Using proof plans to control deduction. Research Paper 185. Department of Artificial Intelligence, University of Edinburgh, 1983.

Warren, D.H.D. WARPLAN – A system for generating plans. Research memo 76. Department of Artificial Intelligence, University of Edinburgh, 1974.

Weyhrauch, R.W. Prolegomena to a theory of mechanized formal reasoning. Artificial Intelligence 13, 133-170, 1980.

USING SYMMETRY FOR THE DERIVATION OF LOGIC PROGRAMS

Anna-Lena Johansson
UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science
Uppsala University
P.O. Box 2059
S - 750 02 Uppsala, Sweden

ABSTRACT

In a programming calculus the formal development of a Horn-clause logic program implies a derivation of program clauses from a set of definitions, of data structures and computable functions, given in full predicate logic. A logic program is composed of a set of program clauses. Each program clause is derived separately from the definitions. Mostly the derivations differ in structure for the different clauses. However, there are cases when two program clauses in a program are similar, except for some small difference, such that they can be transformed into each other. Since the derivations can be lengthy we would like, if possible, to avoid constructing both the derivations. Therefore, after constructing a derivation of a program clause, we would like to be able to answer the question whether there is an analogous program clause, and if so, we would like to know its appearance and the substitution on which to base an application of the substitution rule. In the paper a way to answer this question is discussed.

1: This work is supported by the National Swedish Board for Technical Development (STU)

1 INTRODUCTION

In a programming calculus (see for example Hansson and Tärnlund 1979) the formal development of a Horn-clause logic program Ψ, implies a derivation of program clauses from a set Δ of definitions about data structures and computable functions, given in full predicate logic (see Eriksson, Johansson, and Tärnlund 1983, Hansson 1980, Hogger 1981)

$$\Delta \vdash \Psi.$$

A logic program is composed of a set of assertions (see Kowalski 1982) or program clauses $\psi_1, .., \psi_m \in \Psi$. Each program clause is derived from the definitions, $\delta_1, .., \delta_n \in \Delta$.

$$\delta_1, ..., \delta_n \vdash \psi_1 \wedge ... \wedge \psi_m$$

that is

$$\begin{aligned} \delta_1, ..., \delta_n &\vdash \psi_1 \\ &\cdot \\ &\cdot \\ &\cdot \\ \delta_1, ..., \delta_n &\vdash \psi_m \end{aligned}$$

In general derivations can be quite long and there is always a good idea to be on the lookout for shortcuts. Analogy is a method of exploiting past experience and as such a frequently used method in deductive reasoning. In the area of theorem provers and derivation editors work on the concept of analogy is rare (Bledsoe 1981). An exception is Kling (Kling 1971), who describes a paradigm for reasoning by analogy used by a first-order resolution theorem prover: given a theorem E and the deduction $D_1, ..., D_n \vdash E$ and a conjecture F the analogy between E and F is used to find the set of premises from which the conjecture F is derivable. In the area of program modification and program debugging, analogy has been used both to transform a given program into a new program and to transform an erroneous program into a correct program by finding and using an analogy between two sets of specifications (Dershowitz 1978).

In a situation where we have constructed a derivation of the program clause ψ_i, we would like to know if we can exploit this work to get another of the program clauses by analogy. The program clauses are derived from the same set of premises. Consequently, we are looking for a substitution that transforms program clause ψ_i into a new program clause ψ_j and each definition back to the same definition.

This paper discusses how we can identify the above situation, and furthermore, find the substitution that transforms the derivation of one program clause into a derivation of an analogous program clause.

2 SUBSTITUTIONS

2.1 General definition

Let us assume that we have a derivation of the formula E from the formulas $D_1, ..., D_n$, i.e.

$$D_1, ..., D_n \vdash E.$$

We can substitute formulas for predicate letters throughout the formulas $D_1, ..., D_n$, E, respectively, giving $D_1^*, ..., D_n^*$, E^*. Provided that the substitution is free and all anonymous free variables of the substituting formulas are held constant in the deduction, then according to the formal substitution rule for the predicate calculus (Kleene 1980) we have a deduction of E^* from $D_1^*, ..., D_n^*$, i.e.

$$D_1^*, ..., D_n^* \vdash E^*.$$

Consequently, to convince ourselves that there is a deduction of E^* from $D_1^*, ..., D_n^*$ we need not, although this is possible, construct the derivation itself; instead we rely on the fact that there is a derivation of E from $D_1, ..., D_n$ and a transformation from $D_1, ..., D_n, E$ to $D_1^*, ..., D_n^*, E^*$ respectively. We say that there is an analogy between $D_1, ..., D_n, E$ and $D_1^*, ..., D_n^*, E^*$.

2.2 Special case of substitution for derivation of logic programs

Consider the case when we are deriving a logic program from definitions about data structures and definitions about computable relations or functions, as formulated in the introduction. The definitions correspond to the formulas $D_1, ..., D_n$ in the previous subsection, the program clause corresponds to the consequent E. Each program clause is derived separately from the

definitions. Mostly the derivation tree for the derivation of a program clause ψ_i differs in structure from the derivation tree for a derivation of another program clause ψ_j. A special case arises when there is an analogy between program clauses ψ_i and ψ_i^*, i.e. there is a substitution that transforms ψ_i into ψ_i^*. Our problem is now to find the substitution. Since we want to use the same definitions we want to find a substitution such that, when it is applied to the definitions, it gives back the definitions unaltered except for a reordering of conjunctions, disjunctions, and renaming of bound variables. That is, the substitution has to preserve the structure of the definitions. Every transformed definition δ_i^* is a predicate letter formula in the same predicate letters as the definition δ_i. Consequently, the substitution has to transform δ_i back to itself; it has to be automorphic.

Two important cases where the above properties can be found are

- when a relation is symmetric, i.e. for a predicate letter p we have $\vdash p(a,b) \leftrightarrow p(b,a)$ (see the example of merging two lists in section 3) or
- when two formulas can be consistently interchanged (c.f. principle of duality in projective geometry (Coexter 1969)); if we have a substitution of a predicate letter $r(a,b)$ for a predicate letter $q(a,b)$ we also have the simultaneos substitution of $q(a,b)$ for $r(a,b)$ (see the example of splitting a list in section 3).

By studying the definitions we are able to decide if there exists a substitution that gives an analogous program clause and moreover to conclude that there is a derivation of the analogous clause.

3 SYMMETRIC DEFINITIONS

Let us look at three examples of program derivations. The first example is a program that merges two ordered lists into one ordered list.

The following definition of the merge relation states that the arguments have to be ordered lists and that the third list contains all the elements of the first and the second lists and no more.

Definition δ_1:

$$\forall x \forall y \forall z (merge(x,y,z) \leftrightarrow ordered\_list(x) \wedge ordered\_list(y) \wedge ordered\_list(z) \wedge \forall v(v \in x \leftrightarrow v \in z \vee v \in y))$$

To complete the definition of *merge* we give the definitions of the relations *ordered_list* and $\in$.

The property of being an ordered list can be defined by induction: the empty list is ordered, and the list $u.x$, where u denotes the first element and x denotes the rest of the list, is ordered if, and only if, every element on the list x is greater than or equal to the element u, and the list x is ordered.

Definition δ_2:

$$ordered\_list(\phi)$$

$$\forall u \forall x (ordered\_list(u.x) \leftrightarrow \forall v(v \in x \rightarrow u \leq v) \wedge ordered\_list(x))$$

An element v is defined to be a member of the list $u.x$ if, and only if,

either v is equal to u, or v is member of the list x.

Definition δ_3:

$$\forall v \forall u \forall x (v \in u.x \leftrightarrow v = u \vee v \in x)$$

Finally, greater than or equal is a transitive relation.

Definition δ_4:

$$\forall u \forall v \forall w (u \leq v \wedge v \leq w \rightarrow u \leq w).$$

From the above definitions a recursive program clause can be derived. The program clause asserts that the list $u.x$ is the result of merging the lists $u.x$ and $v.y$ if u is the smallest element of all elements on $u.x$ or $v.y$, that is, if u is less than or equal to v, $u.x$ and $v.y$ are both ordered, and furthermore x has to be the result of merging the lists x and $v.y$.

The program clause, ψ_{merge}: (Universal quantifiers in front of program clauses are omitted.)

$$\begin{aligned} merge(u.x, v.y, u.x) \leftarrow \\ & u \leq v \wedge merge(x, v.y, x) \wedge \\ & ordered\_list(u.x) \wedge \\ & ordered\_list(v.y) \end{aligned}$$

The definitions, $\delta_1, \delta_2, \delta_3$, and δ_4, and the program clause are predicate letter formulas in the predicate letters $merge(a,b,c)$, $ordered\_list(a)$, $a \in b$, $a \leq b$. If we study the definitions, we notice that the definition δ_1 of $merge$ is symmetric with respect to the first and second argument,

$$\delta_1 \vdash \forall x \forall y \forall z (merge(x,y,z) \leftrightarrow merge(y,x,z)).$$

We can, without affecting the definitions, substitute $merge(b,a,c)$ for $merge(a,b,c)$. The substitution in the program clause will consist of substituting $merge(v.y, u.x, u.x)$ for $merge(u.x, v.y, u.x)$ and $merge(v.y, x, x)$ for $merge(x, v.y, x)$. Performing the substitution on the program clause ψ_{merge} gives $\psi_{merge}{}^*$.

The program clause, $\psi_{merge}{}^*$:

$$\begin{aligned} merge(v.y, u.x, u.x) \leftarrow \\ & u \leq v \wedge merge(v.y, x, x) \wedge \\ & ordered\_list(u.x) \wedge \\ & ordered\_list(v.y) \end{aligned}$$

This clause covers the case when the first element of the second list is the smallest element on the two lists to be merged and is congruent to:

$$\begin{aligned} merge(u.x, v.y, v.x) \leftarrow \\ & v \leq u \wedge merge(u.x, y, x) \wedge \\ & ordered\_list(u.x) \wedge \\ & ordered\_list(v.y) \end{aligned}$$

We studied the definitions δ_1, δ_2, δ_3, and δ_4 and found the above substitution that transformed $\delta_1, \delta_2, \delta_3$, and δ_4 to themselves and ψ_{merge} into $\psi_{merge}{}^*$.

If we perform the derivation of ψ_{merge} from $\delta_1, \delta_2, \delta_3, \delta_4$ we can also conclude that $\psi_{merge}{}^*$ is a consequent of the definitions.

That is, if $\delta_1, \delta_2, \delta_3, \delta_4 \vdash \psi_{merge}$
then $\delta_1, \delta_2, \delta_3, \delta_4 \vdash \psi_{merge}{}^*$.

Observe that the program clause, $\psi_{merge}{}^*$ also can be obtained from ψ_{merge} with reference to the replacement theorem instead of the substitution theorem since the relation is symmetric.

Let us look at another example related to the above, but where the

process of finding the analogy is less straightforward, and where we have to use the substitution theorem.

Consider a relation *split* between an element u and three lists x, y, and z, not necessarily ordered, such that x is the combination of y and z, and moreover all elements on y are less than or equal to u, and all element on z are greater than u.

Definition δ_6:

$$\begin{aligned}\forall x\forall u\forall y\forall z(&split(x,u,y,z) \leftrightarrow\\ &list(x) \wedge list(y) \wedge list(z) \wedge\\ &element(u) \wedge\\ &\forall v(v \in x \leftrightarrow v \in y \vee v \in z) \wedge\\ &\forall v(v \in y \rightarrow v \leq u) \wedge\\ &\forall v(v \in z \rightarrow v > u))\end{aligned}$$

The relation $\in$ is already defined in Definition δ_3.

The definition of a list is given recursively; an empty list is a list, and a construction $u.x$ is a list if, and only if, u is an element and x is a list.

Definition δ_8:

$$list(\phi)$$

$$\begin{aligned}\forall u\forall x(&list(u.x) \leftrightarrow\\ &element(u) \wedge list(x))\end{aligned}$$

From the definition of *split* and the definitions of $\in$ and *list* a program clause can be derived; if the first element v on the first list is less than the discriminating element u and the result of splitting the rest of the first list x are the lists y and z then, v has to be the first element on the list in the third argument place.

Program clause, ψ_{split}:

$$\begin{aligned}&split(v.x,u,v.y,z) \leftarrow\\ &\quad v \leq u \wedge split(x,u,y,z)\end{aligned}$$

The definition of split is symmetric with respect to the third and fourth argument if we interchange $a \leq b$ and $a > b$.

Program clause, $\psi_{split}{}^*$:

$$\begin{aligned}&split(v.x,u,z,v.y) \leftarrow\\ &\quad v > u \wedge split(x,u,z,y)\end{aligned}$$

Let us look at yet one more example where the symmetry is not in the definition at the top of the hierarchy of definitions. We want to derive a program that inserts elements into an ordered binary tree, the result also being an ordered binary tree.

The relation *insert* is a relation between two ordered binary trees and a label. The second binary tree w' is equal to the first w with the label v inserted at the appropriate place.

Definition δ_7:

$$\begin{aligned}\forall w\forall v\forall w'(&insert(w,v,w') \leftrightarrow\\ &ordered\_binary\_tree(w) \wedge\\ &label(v) \wedge\\ &ordered\_binary\_tree(w') \wedge\\ &\forall v'(v' \in w' \leftrightarrow v' = v \vee v' \in w))\end{aligned}$$

The property of being an ordered binary tree can be defined by induction: an empty binary tree is ordered, and a binary tree $t(x,u,y)$ is ordered if, and only if, its subtrees are ordered, and all elements on the left subtree x are less than the root u, and all elements on the right subtree y are greater than the root of the tree.

Definition δ_8:

$$ordered\_binary\_tree(\phi)$$

$$\forall x \forall u \forall y(ordered\_binary\_tree(t(x,u,y)) \leftrightarrow label(u) \wedge ordered\_binary\_tree(x) \wedge ordered\_binary\_tree(y) \wedge \forall v(v \in x \rightarrow v < u) \wedge \forall v(v \in y \rightarrow v > u))$$

An element v is member of the binary tree $t(x,u,y)$ if, and only if, it is equal to the root u, or is member of one of the subtrees x and y.

Definition δ_9:

$$\forall v \forall x \forall u \forall y(v \in t(x,u,y) \leftrightarrow v = u \vee v \in x \vee v \in y)$$

From the above definitions we can derive a program clause for a leaf insertion program stating that the tree $t(x',u,y)$ is the result of inserting v into $t(x,u,y)$, if v is a label and $t(x,u,y)$ is an ordered binary tree, and v is less than the root u, and the result of inserting v in the left subtree x is x'.

The derived program clause ψ_{insert}

$$insert(t(x,u,y),v,t(x',u,y)) \leftarrow v < u \wedge label(v) \wedge ordered\_binary\_tree(t(x,u,y)) \wedge insert(x,v,x')$$

Applications of binary trees are often symmetric with respect to the left and the right subtrees (Knuth 1975). The membership-relation in definition δ_9 is symmetric

$$\delta_9 \vdash \forall v \forall x \forall u \forall y(v \in t(x,u,y) \leftrightarrow v \in t(y,u,x)).$$

If we reflect the predicates less than and greater than into each other the definition of *ordered_binary_tree* for a non-empty tree, is also symmetric with respect to the left and right subtrees. The definition of *insert* is given in terms of *ordered_binary_tree* and $\in$.

Instantiating the definition of *insert* with non-empty ordered binary trees gives

$$insert(t(x,u,y),v,t(x',u',y')) \leftrightarrow ordered\_binary\_tree(t(x,u,y)) \wedge label(v) \wedge ordered\_binary\_tree(t(x',u',y')) \wedge \forall v'(v' \in t(x',u',y') \leftrightarrow v' = v \vee v' \in t(x,u,y)).$$

Application of the substitution of $ordered\_binary\_tree(t(c,b,a))$ for $ordered\_binary\_tree(t(a,b,c))$ carries over to the relation *insert*. We have to substitute $insert(t(a,b,c),d,t(e,f,g))$ for $insert(t(c,b,a),\ d,t(g,f,e))$ as well.

The transformed program clause ψ_{insert}^* inserts the new label into the right subtree.

$$insert(t(y,u,x),v,t(y,u,x')) \leftarrow v > u \wedge label(v) \wedge ordered\_binary\_tree(t(y,u,x)) \wedge insert(x,v,x')$$

From the specification of *insert* we can also derive a root insertion program.

The program clause ψ_{root_insert}:

$$insert(t(x,u,y),v,t(x',v,t(x'',u,y))) \leftarrow v < u \wedge label(v) \wedge ordered\_binary\_tree(t(x,u,y)) \wedge insert(x,v,t(x',v,x''))$$

We can obtain the dual case using the same methods as above.

$$insert(t(y,u,z),v,t(t(y,u,z''),v,z')) \leftarrow$$
$$v > u \wedge$$
$$label(v) \wedge$$
$$ordered\_binary\_tree(t(y,u,z)) \wedge$$
$$insert(z,v,t(z'',v,z'))$$

4 HOW TO FIND THE SUBSTITUTION

Let us look at the example of splitting a list described in the previous section. Our task is to find a substitution satisfying the properties mentioned in subsection 2.2. Our starting point is the derivation of ψ_{split} from the definitions δ_3, δ_5, and δ_6. We normalize the derivation (Gentzen 1934, Prawitz 1965, Stålmarck 1983), in order to make sure that the candidates for substitution are predicate letters. Therefore, if the least complex formulas used in the normalized derivation (minimum formulas) contain a logical constant we add a new definition with the formula as definiens and substitute throughout the derivation. For example if $q(a,b) \wedge r(c)$ is a minimum formula then we define $\forall z \forall y \forall z(p(z,y,z) \leftrightarrow q(z,y) \wedge r(z))$ and consequently replace $q(a,b) \wedge r(c)$ with $p(a,b,c)$.

The definitions δ_3, δ_5 and δ_6 are studied, one by one, in order to find the right substitution. Let us concentrate on the definition δ_5 describing the relation *split*. We identify the predicate letter in the definiens as $list(a)$, $element(a)$, $a \in b$, $a \leq b$, and $a > b$.

Each occurrence of a predicate letter is inserted into an equivalence class according to the sequence of rule applications that has to be performed to decide that the occurrence is a subformula of the definiens.

The rules deciding that a formula is a subformula are the following:

1. A is a subformula of A.
2. If $A_1 \wedge ... \wedge A_n$ is a subformula of A, then so are A_i, $1 \leq i \leq n$.
3. If $A_1 \vee ... \vee A_n$ is a subformula of A, then so are A_i, $1 \leq i \leq n$.
4. If $B \rightarrow C$ is a subformula of A, then so is B.
5. If $B \rightarrow C$ is a subformula of A, then so is C.
6. If $B \leftrightarrow C$ is a subformula of A, then so are B and C.
7. If $\neg B$ is a subformula of A, then so is B.
8. If $\forall x B$ is a subformula of A, then so is B^x_t.
9. If $\exists x B$ is a subformula of A, then so is B^x_t.

From the definition of split

$$\forall x \forall u \forall y \forall z(split(x,u,y,z) \leftrightarrow$$
$$list(x) \wedge list(y) \wedge list(z) \wedge$$
$$element(u) \wedge$$
$$\forall v(v \in x \leftrightarrow v \in y \vee v \in z) \wedge$$
$$\forall v(v \in y \rightarrow v \leq u) \wedge$$
$$\forall v(v \in z \rightarrow v > u))$$

we obtain the following equivalence classes of subformulas

1. $[list(x), list(y), list(z), element(u)]$ from the sequence of rule applications 2,1
2. $[v \in x]$ from the sequence 2,8,6,1

3. $[v \in y, v \in z]$ from the sequence 2,8,6,3,1

4. $[v \in y, v \in z]$ from the sequence 2,8,4,1

5. $[v \leq u, v > u]$ from the sequence 2,8,5,1

To find the substitution we study the equivalence classes one by one. All the members in an equivalence class can be used in the same way in a derivation, they are interchangable keeping the derivation structure. We try to make pairs within the eqivalence class, identify the difference in the pairs and study the result of applying the transformation to the rest of the equivalence classes and the definiens.

If we first convert the definiens into a normal form the set of rules for deciding the subformula property reduces and so does also the number of equivalence classes that can be obtained. If, in the definition of *split* the subformula $\forall v(v \in y \rightarrow v \leq u)$ was formulated as $\forall v(\neg(v \in y) \vee v \leq u)$ and the rest of the definiens unchanged, then we should not be able to find the right substitution with the suggested method without converting into a normal form. The definitions in this paper are presented on a form where substitution can be found without conversion.

Let us look at the classes above, class 2 has only one member and can therefore be discarded, we can make no pair there. The two equal classes 3 and 4 can be reduced to one. The difference in the formulas in equivalence class 1 is the argument to *list*. Interchanging x and y or x and z in the classes gives a new class instead of class 3 but interchanging y and z gives back the old classes. The change of y and z corresponds to changing the third and fourth argument of the relation *split*. The elements in equivalence class 3 differ again in y and z. The elements in eqivalence class 5 does not occur in any of the other classes. Interchanging the two elements gives the same collection of classes as before.

The substitutions then are

- $split(a, b, d, c)$ for $split(a, b, c, d)$

- $a \leq b$ for $a > b$ and $a > b$ for $a \leq b$.

Substitution in the definition of *split* results in a definition that is equivalent to itself.

$$\begin{aligned}\forall x \forall u \forall y \forall z(split(x,u,z,y) \leftrightarrow\ & list(x) \wedge list(z) \wedge list(y) \wedge \\ & element(u) \wedge \\ & \forall v(v \in x \leftrightarrow v \in z \vee v \in y) \wedge \\ & \forall v(v \in x \rightarrow v > u) \wedge \\ & \forall v(v \in y \rightarrow v \leq u))\end{aligned}$$

The other definitions used in the derivation of the program clause do not contain the predicate letters $split(a, b, c, d)$, $a \leq b$, or $a > b$ and are therefore not affected.

5 CONCLUSIONS

A method for deciding if there are analogous program clauses to be considered when deriving a logic program is proposed. The proposal is based on the notion of symmetry and reflection in predicates. The handling of symmetric data structures is not straightforward and needs further investigation. The amount of work for deriving a logic program when there is analogy between the program clauses can be reduced using this method.

ACKNOWLEDGMENT

I wish to thank Åke Hansson, who spent much time reading drafts of this paper and suggesting improvements.

REFERENCES

Bledsoe, W. W. Non-resolution theorem proving, Readings in Artificial Intelligence, ed. Webber, B. L. and Nilsson, N. J., Tioga Publishing Company, Californien, 1981.

Coxeter, H. S. M. Introduction to Geometry, John Wiley and Sons, second edition, 1969.

Dershowitz, N. The Evoluation of Programs, Ph. D. Thesis, Weizmann Institute of Science, Israel, 1978.

Eriksson, A., Johansson, A-L and Tärnlund, S-Å Towards a Derivation Editor, Logic Programming and its Applications, ed D. Warren and M. van Caneghem, 1983.

Gentzen, G. Untersuchungen über das logische Schliessen, Matematische Zeitschrift vol. 39, 1934-1935.

Hansson, Å. A Formal Development of Programs, Ph.D. Thesis, Department of Information Processing and Computer Science, The Royal Institute of Technology and The University of Stockholm, 1980.

Hansson, Å. and Tärnlund, S-Å. A Natural Programming Calculus, Proc. IJCAI-6, Tokyo Japan, 1979.

Hogger, C. J. Derivation of Logic Programs, JACM, Vol 28, No 2, April 1981, pp 372-392, 1981.

Kleene, S.C. Introduction to Metamathematics, North-Holland, Amsterdam, Eight reprint, 1980.

Kling, R.E. A Paradigm for Reasoning by Analogy, Artificial Intelligence 2, p 147-178, 1971.

Knuth, D.E. The Art of Computer Programming, Volume 1, Fundamental Algorithms, Addison-Wesley Publishing Company, Second Edition, 1975.

Kowalski, R. A. Logic as a Computer Language, in Logic Programming, Academic Press, 1982, eds. Clark, K.L. and Tärnlund, S.Å..

Prawitz, D. Natural Deduction. A Proof-Theoretical Study, Ph. D. Thesis, Almqvist and Wiksell, Stockholm, 1965.

Stålmarck, G. Strong normalization for complete 1st order classical natural deduction, Stockholm, 1983.

A MODEL THEORY OF LOGIC PROGRAMMING METHODOLOGY

Huaimin Sun and Liguo Wang
Computer Science Department
Beijing Institute of Aeronautics
and Astronautics
Beijing China

ABSTRACT

A new version of the first order language for mechanical theorem proving is axiomatized. It is called "Subgoal Deduction Language"(SDL) and is used as a metalanguage for specification and derivation of logic programs as well as for representation of the knowledge necessary for program reasoning. A many sorted relation system is defined as the semantic interpretation of SDL and logic programs. An example of an automatic derivation of a logic program from its specification is given.

1 INTRODUCTION

In recent years many brilliant works in the field of derivation of logic programs have been done(See references from 2 to 5, for example). In all these works deduction has played a central role. However, an auto-programming system must has the ability to derive the specifications of subprocedures from the specifications of the main program. In order to solve this problem, we have axiomatized a goal oriented deduction system SDL and implemented it in Prolog language on our microcomputer BCM-3 as an experimental auto-programming system. The semantical interpretation of SDL based on the notations of model theory is given. The experimental results show that the SDL can really draw the necessary specifications of the subprocedures. This behaviour is illustrated by an example in this paper.

2 MANY SORTED SEMANTICAL SYSTEMS

A many sorted semantical system is defined as

$\Sigma = \langle D_1, \ldots, D_t, R, \gamma_1, \ldots \gamma_g, \ldots, f_1, \ldots f_h, B \rangle$

where $D_i (1 \leq i \leq t)$ is the object of the i-th sort which is a well-founded set and can be specified by the 3-tuples

$\langle \perp, \leq, succ \rangle$

where $\perp$, $\leq$, succ are respectively the minmum element, the partial order and the function of successor. R is a finite set of g relational symbols r1,...,rg. Each $r_i \in R$ is a mapping from $D_{i1} \times D_{i2} \times, \ldots, \times D_{in}$ to B = {true, false}. We call $\langle D_{i1}, \ldots, D_{in} \rangle$ the type of r_i. For every $r_i \in R$, there is a set r_i in Σ

$r_i = \{(d_{i1}, \ldots, d_{in}) \mid r_i \in R, \langle D_{i1}, \ldots, D_{in} \rangle$ is the type of r_i, $d_{ij} \in D_{ij}$ $(1 \leq j \leq n)$, $r_i(d_{i1}, \ldots, d_{in})$ is true$\}$

Each functional symbol fj $(1 \leq j \leq h)$ is a mapping from $D_{j1}, \ldots, D_{jm}$ to D_j. The type of fj is re-

presented by $\langle Dj1,\ldots,Djm, Dj\rangle$.

Using the relational symbols, functional symbols, constants (as the names of the objects of Di) together with variables, quantifiers and logical connectives we can write the well-formed semantical formulas (swff) over Σ similar to the wff of standard first order predicate caculus. The only difference between wff and swff is that the predicate symbols in the former is replaced by the relational symbols in the latter. The truth value of a swff is defined by $\underline{r1},\ldots,\underline{rg}$ in Σ. The set of all the true swffs over Σ is denoted by "diagΣ".

Now we can give a formalism of the task of programming as follows.

Definition 2.1

Let P be a programming task with m input variables and n-m output variables, r1(X1,...,Xm, Zm+1,...,Zn) be the expected input-output relation defined on the abstract data structures D1,..., Dt. r2,...,rg be other known relations defined on D1,...,Dt. f1, ...,fh be known functions defined on D1,...,Dt. Then the semantical system

$$\Sigma_P=\langle D1,\ldots,Dt, \{r2,\ldots,rj\},\underline{r2},\ldots,\underline{rg}, f1,\ldots,fh, B\rangle$$

is called the knowledge background of P, r1 the goal of P.

Let Σ_P be a knowledge background of P, then the knowledge necessary for the programming task can represented as a finite subset K of diagΣ_P. If we can write a swff $S \in \text{diag}\,\Sigma_P$ satisfying

$$\{(x1,\ldots,xm,zm+1,\ldots,zn)\mid S(x1,\ldots,xm,zm+1,\ldots,zn)\}=\{(x1,\ldots,xm,zm+1,\ldots,zn)\mid r1(x1,\ldots,xm,\ldots,zn)\}$$

then the swff S is called the specification of P.

Example 2.1

Suppose it is required to design a logic program "arrange(L, T)" which constructs a ordered binary tree T with all the integers in a given list L as its nodes. The knowledge background of P is

$$\Sigma_P=\langle \text{INT, INT-LIST, TREE, R, } \underline{le}, \underline{gt}, \underline{member}, \underline{node}, \underline{order}, \text{cons, car, } \underline{cdr}\rangle$$

where INT, INT-LIST,TREE are respectively the relation "less than or equal to", "greater than". $\underline{le}=\{(1,1), (1,2),\ldots\}$. $\underline{gt}=\{(2,1), (3,2),\ldots\}$. relation member (X,L) denotes that the integer X is an element of the int-list L. $\underline{member}=\{\ldots, (2,(2,3,4)),\ldots\}$. node(X, T) denotes that the integer X is a node of a binary tree. $\underline{node}=\{(1,t(nil, 1,nil)),\ldots\}$. order(T) denotes that the binary tree T is sorted. cons, car, cdr are known functions.

The specification of the program can be expressed with the following swff $\in$ diagΣ_P

$$\text{arrange}(L,T) \triangleq (\forall N):N \in \text{INT } (\forall L):L\in\text{INT-LIST } (\exists T):T\in\text{TREE}((\text{member}(N,L)\leftrightarrow\text{node}(N,T))\wedge \text{order}(T)) \qquad \text{(def.1)}$$

Suppose that the relations member, node, order are not the build-in relations of the auto-programming system, then they need to be defined with the swff's.

The definition of "binary tree" is as follows

$$\text{tree}(nil) \qquad \text{(def.2.1)}$$

$$(\forall X):X\in\text{INT } (\forall T1):T1\in\text{TREE } (\forall T2):T2\in\text{TREE } (\text{tree}(T1)\wedge\text{tree}(T2)\leftarrow\text{tree}(t(T1,X,T2))) \qquad \text{(def.2.2)}$$

An ordered binary tree is defined as follows

order(nil) (def.3.1)

(∀T1):T1∈TREE (∀T2):T2∈TREE(∀X):
1∈INT (∀N):N∈INT((node(N, T1)→N≤
I)∧(node(N, T2)→N>X)∧order(T1)∧
order(T2)→order(t(T1, X, T2)))
(def.3.2)

where node(N,T) is defined as

(∀N):N∈INT ¬node(N,nil)
(def.4.1)

(∀N):N∈INT(∀X):X∈INT(∀T1):T1∈
TREE(∀T2):T2∈TREE(node(N,t(T1,X,
T2))↔N=X ∨ node(N,T1) ∨ node(N,
T2)) (def.4.2)

The relation member(X,L) is defined as

(∀N):N ∈ INT ¬member(N,nil)
(def.5.1)

(∀N):N ∈ INT(∀X):X ∈ INT(∀L):L
∈LIST(member(N,X·L)↔N=X ∨ member
(N,L)) (def.5.2)

Thus K={(def.2),(def.3),(def.4),(def.5)} is the representation of the knowledge over Σ_P, necessary for the programming task of "arrange(L,T)".

3 THE SUBGOAL DEDUCTION LANGUAGE SDL

3.1 Sentences

SDL consists of sentences. A sentence is in fact a first order predicate formula in the form:

p1,...,pm→q1,...,qn

where pi ($1 \le i \le m$), qj ($1 \le j \le n$) are all literals (atomic predicate formulas or negative of them) with terms consisting of in the same way as in standard predicate calculus (we shall use the upper case letters to denote variables, lower case letters to denote constants and functions, lower case letters with subscript "e" to denote Skolem functions). The commas on the left hand side and right hand side of "→" are read respectively as "and" and "or". All the variables in a sentence are thought to be as variables bounded by universal quantifiers. We take p1,...,pi-1, pi+1, ..., pm→¬pi, q1,..., qm (¬qj,p1, ..., pm→q1,...,qj-1, qj+1,..., qn) as the equivalent form of p1, ..., pm→q1,..., qn. In a deduction they can be transformed to each other. The sentences "→q", "p→ " can be read as "true→q", "p→false", which are equivalent to q, ¬p respectively.

3.2 Inference Rules System

3.2.1 S-deduction

Let K be a finite set of SDL sentences, α be a sentence ∈ SDL. If α can be deduced by the following inference rules from K, then we may say α is a valid consequence of K, denoted by

$$K \vdash_S \alpha$$

We call such a proof a "S-deduction of α from K".

In a S-deduction the sentence α should be first transformed according to the following rules.

(1) The variable X in α is replaced by a special constant symbol xu, which is called a formal constant. It must be distinguished from any constant in K or other formal constants in α.

(2) The Skolem function fe in α is replaced by a function symbol Fe which is called a Skolem function variable. We use the upper case letters with a subscript "e" to denote it. It must be distinguished from any Skolem function symbol in K.

The sentence obtained by the above two rules from α is called the goal form of α, denoted by "α'".

Note that the rule (1) is simply an application of the generalization rule in the standard predicate calculus. For if we want to prove $(\forall X): X \in Di\ \alpha(X)$, all we need to do is to prove $\alpha(xu)$ for a arbitrary chosen object name $xu \in Di$.

The meaning of rule (2) is that since we want to prove $(\exists X): X \in Di\ \alpha(X)$, we may choose any object $x \in Di$ to replace X. If $\alpha(x)$ is proved, so is $(\exists X)\alpha(X)$.

It is obvious that we should have $K \vdash_S \alpha$ iff $K \vdash_S \alpha'$.

3.2.2 Valid Substitution

A valid substitution σ of a sentence is a substitution according to the following rules

(1) Any term can be substituted for a variable.

(2) A Skolem function or a known function symbol can be substituted for a Skolem function variable provided that the argument set of the former is a subset of the argument set of the latter. We view a constant as a 0-ary function.

If there is a valid substitution σ which makes $\alpha_1(t1,\ldots,tn) = \alpha_2(t1',\ldots,tn')$, then it is called the valid unification between these two sentences.

3.2.3 Inference Rules

In the following we use the schema

$$\frac{\alpha'}{\beta'}\ \uparrow \gamma$$

to denote that under the condition γ we have $K \vdash_S \alpha'$ if $K \vdash_S \beta'$.

a. Subgoal Rules

(1)

$$\frac{(\alpha \rightarrow \beta 1, \beta 2)'}{(\alpha \rightarrow p1, \beta 2)'\sigma}\ \uparrow \begin{array}{l}(p1,\ldots,pm \rightarrow \beta 1) \in K \\ \sigma \text{ is a valid unification between } \beta 1' \text{ and } \beta 1\end{array}$$
$$\ldots$$
$$(\alpha \rightarrow pm, \beta 2)'\sigma$$

(2)

$$\frac{(\alpha 1, \alpha 2 \rightarrow \beta)'}{(g1, \alpha 2 \rightarrow \beta)'\sigma}\ \uparrow \begin{array}{l}(\alpha 1 \rightarrow q1,\ldots, qm) \in K \\ \sigma \text{ is a valid unification between } \alpha 1' \text{ and } \alpha 1\end{array}$$
$$\ldots$$
$$(gm, \alpha 2 \rightarrow \beta)'\sigma$$

(3)

$$\frac{\alpha(t1)'}{\alpha(t2)'}\quad (t1=t2) \in K$$

(4)

$$\frac{t1=t2 \rightarrow \alpha(t1)'}{\alpha(t2)'}$$

A sentence $\alpha = p1,\ldots,pm \rightarrow q1,\ldots,qn$ is a valid consequence of K if

(i) There is a valid unification between a sentence in K and α' or some literals $qi',\ldots,qj'$ ($1 \leq i \leq j \leq n$).

(ii) All the subgoals of α' deduced by subgoal rules are valid consequence of K.

b. Conditional Rule

Let $\alpha_1', \ldots, \alpha_i', \ldots, \alpha_m'$ be subgoals of α' and α_i' include only constants and known functions as its terms. If $\alpha_1', \ldots, \alpha_{i-1}', \alpha_{i+1}', \ldots, \alpha_m'$ are all valid

consequences of K ∪ {α_i'} and α' is a valid consequence of K ∪ {$\neg\alpha_i'$}, then α' is a valid consequence of K.

We give a simple example of the application of the inference rules. The problem is extracted from [6].

Example 3.1

Suppose we want to prove (∃X) happy(X) from K={(∀X)(∀Y)(employs (X,Y) ∧ working(Y)→happy(X)), (∀X)(playing(X)→happy(X)), playing(bob) ∨ working(bob), employs (john,bob)}.

A S-deduction is as follows
The set K includes

employs(X,Y), working(Y) → happy(X) (K.1)

playing(X)→ happy(X) (K.2)

¬working(bob)→ playing(bob) (K.3)

employs(john,bob) (K.4)

The goal is happy(Xe). It can be matched with (K.1) according to the subgoal rule (1) with a substitution ⟨ Xe/X ⟩ and thus deduce two subgoals:

employs(Xe,Y) (Goal 1.1)

working(Y) (Goal 1.2)

(Goal 1.1) can be easily proved with substitution ⟨ john/Xe, bob/Y⟩, but then (Goal 1.2) becomes to working(bob), which can not be proved by K. However, it can be easily proved by K∪{working(bob)}. So according to the conditional rule all we need to do is to prove K∪{¬working(bob)}⊢ happy(Xe), which can be proved with substitution ⟨bob/Xe⟩, using subgoal rule (1) and (K.2), (K.3). So we know that if Bob is working, John is happy, otherwise Bob is happy. Thus (∃X)happy(X) is proved.

4 THE SEMANTICS OF SDL

Definition 4.1

Let Σ=⟨D1,...,DT, R, r1,..., rg, f1,...,fh, B⟩ be a semantical system. Let K be a finite subset of SDL. A valuation μ_K of K is a mapping from K to Σ which

(1) associates an object set Di of Σ with each universe of K.

(2) associates an object of Σ in Di with each constant of K in the image universe of the objects domain Di.

(3) associates a relation symbol with each predicate of K.

(4) associates a mapping Dj1, ..., Djm → Dj with each m-ary function or m-ary Skolem function occuring in K.

(5) associates truth value "true" with each literal p(t1,..., tn) in K iff (t1,...,tn) ∈ r, where r is the image relation of p.

Definition 4.2

A valuation μ_K is a model of K if it associates "true" to every sentence of K.

Definition 4.3

Let α be a SDL sentence, consisting of literals occuring in K. A valuation $\mu_{K\alpha}$ of α according to μ_K is a mapping from α to Σ where:

(1) The images of its predicates are the same as in μ_K.

(2) If t is a term which occurs both in α and K, then its image is the same as in μ_K.

(3) If t is a Skolem function with type ⟨ Di1,...,Dim, Di⟩ which occurs only in α, then associates a mapping Di1 X,...,XDim → Di with it.

We assume that all constants or functions which occur in α also occur in K.

Definition 4.4

Let K be a subset of SDL and α be a SDL sentence consisting of literals occuring in K. We say α is a semantical conclusion of K, denoted by "$K \models \alpha$ ", if for any model $\mathcal{M}_K$ of K there is a $\mathcal{M}_{K\alpha}$ which associates α with "true".

We have the following theorems, the proofs of which is omitted in this paper.

Theorem 4.1 (The Soundness of Valid Unification)

Let $K \vdash_S \alpha$ be a S-deduction and α' be the goal form of α. If there is a valid unification between α' and a sentence S in K, then $K \models \alpha$.

Theorem 4.2 (The Soundness of Subgoal Rules)

Let $K \vdash_S \alpha$ be a S-deduction and α' be the goal form of α. Let $\beta 1'$, ..., $\beta m'$ be the subgoals deduced from α' according to subgoal rules 3.2.3 a. If for every $\beta i'$ $(1 \le i \le m)$ there is a valid unification between $\beta i'$ and a sentence Si in K, then $K \models \alpha$.

Theorem 4.3 (The Soundness of S-deduction)

$K \models \alpha$ if $K \vdash_S \alpha$

Theorem 4.4 (The Soundness of Conditional Rule)

Let $K \vdash_S \alpha$ be a S-deduction, α' be the goal form of α. Let $\alpha 1'$, ..., $\alpha i'$, ..., $\alpha m'$ be subgoals of α' and $\alpha i'$ includes only constants and known functions as its terms. Then we have $K \models \alpha$ if $K \cup \{\alpha i'\} \vdash_S \alpha 1'$ & ... & $\alpha i-1'$ & $\alpha i+1'$ & ... & $\alpha m'$ and $K \cup \{\neg \alpha i'\} \vdash_S \alpha'$.

Note that the S-deduction system is not complete. However, using generalization rule and deduction theorem in standard predicate calculus, any provable wff can be transformed into a equivalent form which can be proved by S-deduction, but we shall not discuss this problem here.

5 AN EXAMPLE OF AUTOMATIC DERIVATION OF LOGIC PROGRAMS

We have written a prolog program which implemanted SDL together with a structural induction mechanism as an experimental programming system. Here we give an example to show how the system derives automatically a logic program from the knowledge background and specifications given in the example 2.1 in section 2.

The specification and knowledge given in example 2.1 can be expressed in SDL sentences as follows

Specification

$[\![1 \in \text{INT-LIST}, fe(1) \in \text{TREE}]\!]$:arrange(1,fe(1))

$N \in 1 \rightarrow node(N,fe(1))$ (Spec.1)

$node(N,fe(1)) \rightarrow N \in 1$ (Spec.2)

order(fe(1)) (Spec.3)

Knowledge (We write "$\alpha \leftrightarrow \beta$" instead of two sentences "$\alpha \rightarrow \beta$" and "$\beta \rightarrow \alpha$" for covenience.)

tree(nil) (Theorem.1)

$int(X) \wedge tree(T1) \wedge tree(T2) \leftrightarrow tree(t(T1,X,T2))$ (Theorem.2)

$\neg node(N,nil)$ (Theorem.3)

$N=X \vee node(N,T1) \vee node(N,T2) \leftrightarrow node(N,t(T1,X,T2))$ (Theorem.4)

order(nil) (Theorem.5)

order(t(T1,X,T2)) $\overset{\mathrm{d}}{=}$

node(N,T1)→N≤X (Theorem.6)
&
node(N,T2)→N ≥ X (Theorem.7)
&
order(T1) (Theorem.8)
&
order(T2) (Theorem.9)

¬(N ∈ [ ]) (Theorem.10)

N ∉ X·L↔N=X ∨ N ∈ L
(Theorem.11)

Let K= {(Theorem.1),...,(Theorem.11)}, the program satisfying (Spec.1) to (Spec.3) can be derived by the following S-deduction

K⊢

nu ∈ l→node(nu,Fe(l))
(Goal.1)
&
node(nu,Fe(l))→nu ∈ l
(Goal.2)
&
order(Fe(l)) (Goal.3)

Step 1 Derivation of induction base (induction on the length of the interger list)

Let l=[ ], (Goal.1) to (Goal.3) can be easily proved by the valid unification

$$\sigma_1 = \langle nil/Fe([\ ]) \rangle$$

So the following solution is obtained.

f-arrange([ ])=nil (S.1)

Step 2 Inductive inference

The system assums that the given input can be expressed as "x·l" and for any integer list L shorter than x·l, f-arrange(L) is a known function (which is writter as f-a(L) in the follows for convenience). Thus the system obtains the following induction hypotheses

L<x·l, N ∈ L→node(N,f-a(L)) (H.1)

L<x·l,node(N,f-a(L))→N ∈ L (H.2)

L<x·l→ order(f-a(L)) (H.3)

The system uses K ∪ {(H.1), (H.2), (H.3)} as hypotheses set and tries to prove (Goal.1) to (Goal.3) again, substituting x·l for l, as follows

nu ∈ x·l→node(nu,Fe(x·l)) (Goal.1')

node(nu,Fe(x·l))→nu ∈ x·l (Goal.2')

order(Fe(x·l)) (Goal.3')

Then the system tries to prove these goals. In order to restrict the number of wrong search paths, the proving process of inductive inference in our experimental system is divided into two stages. In stage 1 the system uses its knoledge in K and inference rules to transform (Goal.1') to (Goal.3') into subgoals expressed by x and l separately. In stage 2 the system tries to find a solution for the program by proving these subgoals, using inference rules and induction hypotheses only. If it succeeds, the derivation stops, otherwise the system will summerize those unproved subgoals as specifications of some desired subprocedures and tries to prove them. In order to save space, we will not give the whole derivation here, but only give the proving process of (Goal.1') as an illustration. Readers who are interested in the details can obtain the experimental records from our institute. The proving of (Goal.1') is shown in Fig.1.

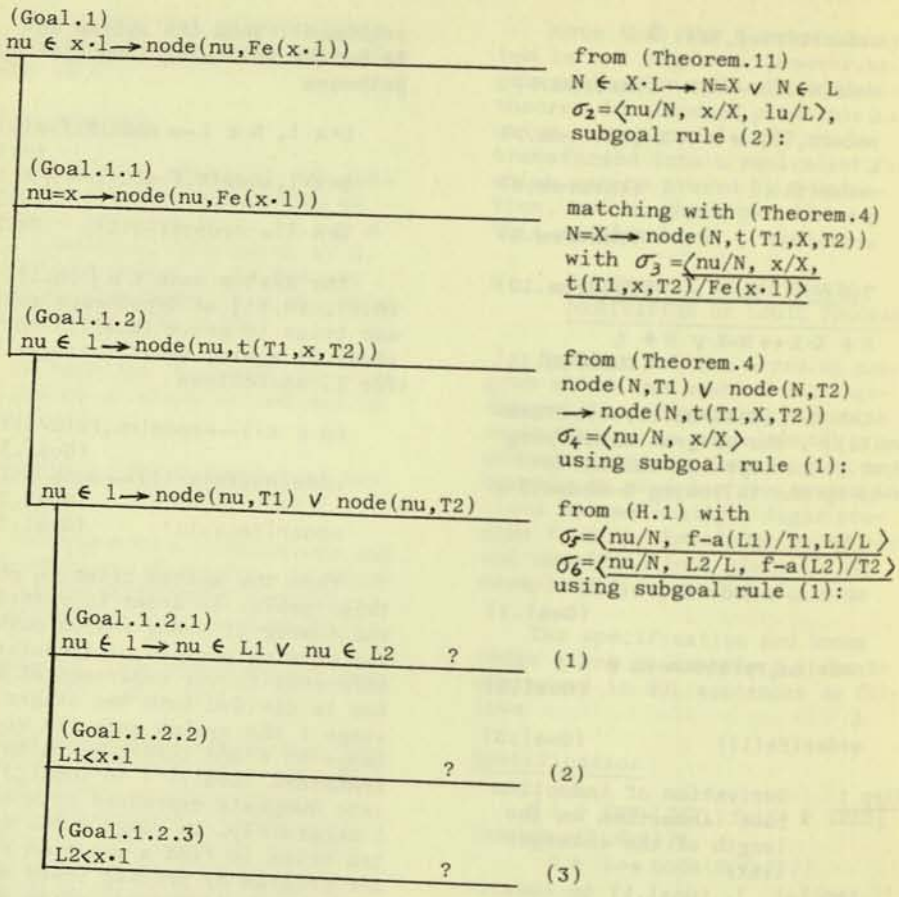


Fig.1

Note that the substitution σ_3, σ_5, σ_6 in Fig.1 are underlined, because they show that the solution of Fe(x·1) may take the form

Fe(x·1)=t(T1,x,T2)

T1=f-a(L1)

T2=f-a(L2)

But the solution are not complete, since there are variables L1 and L2 occur in them. They have to be constructed further in the proof.

The unproved subgoals (1), (2) and (3) in Fig.1 are summerized as part of specifications of procedures f1e(l,x)=L1 and f2e(l,x)=L2. In further proving process of (Goal.2') and (Goal.3') it is showed that the uncompleted solution form is suitable and other specifications of subprocedures are obtained in a similiar way. Thus the system obtains the following intermediate solution

```
f1e(l,x)=L1

f2e(l,x)=L2

f-arrange(L1)=T1, f-arrange(L2)
=T2→f-arrange(x·l)=t(T1,x,
T2)
```

(S.2)

and the specifications of subprocedures are as follows

Specification

⟦l,f1e(l,x),f2e(l,x),∈ INT-LIST,N,x ∈ INT⟧:

N ∈ l→N ∈ f1e(l,x) ∨ N ∈ f2e(l,x) (Spec.1')

N ∈ f1e(l,x)→ N ∈ l (Spec.2')

N ∈ f2e(l,x)→ N ∈ l (Spec.3')

N ∈ f1e(l,x)→N≤x (Spec.4')

N ∈ f2e(l,x)→N>x (Spec.5')

f1e(l,x)≺x·l (Spec.6')

f2e(l,x)≺x·l (Spec.7')

The specification shows that the task of the sub-procedures is to partite the input integer list l into two lists: f1e(l,x) and f2e(l,x). The former includes all the integers in l which are less than or equal to x and the latter all the integers in l which are greater than x. So we see that the SDL can deduce automatically the specification for the desired sub-procedures in a very natural way.

After derivation of the sub-procedure, the system obtains the following solution (in which we write "f1-partition" and "f2-partition" instead of "f1e" and "f2e" to explicate the meaning of them).

```
f1-partition([ ],x)=[ ].

f2-partition([ ],x)=[ ].

n≤x→f1-partition(n·l,x)
    =n·f1-partition(l,x).

n≤x→ f2-partition(n·l,x)
    =f2-partition(l,x).

n>x→ f1-partition(n·l,x)
    =f1-partition(l,x).

n>x→ f2-partition(n·l,x)
    =n·f2-partition(l,x).
```

(S.3)

From solutions (S.1) to (S.3), a Prolog program in standard form can be easily obtained.

REFERENCES

[1] Sun, H., Wang, L. An Auto-Programming Technique Using Subgoal Deduction and Structural Induction. Chinese Journal of Computers Vol.6, No.6. 1983.

[2] Clark, K.L., Tärnlund, S.Å. A First Order Theory of Data and Programs.Information Processing 77 IFIP.

[3] Sickel, S. Specification and Derivation of Programs. Theoretical Foundations of Programming Methodology.Lecture Notes of an International Summer School. D. Reidel Publishing Company.1982.

[4] Hansson, L. Tärnlund, S.Å. A Natural Programming Calculus. Proc. JCA-6, Tokyo, Japan. 1979.

[5] Hogger, C.
Derivation of Logic Programs.
JACM 28. No 2. 1981.

[6] Kowalski, R.A.
Logic for Problem Solving.
North Holland Inc. 1979.

A UNIFIED TREATMENT OF RESOLUTION STRATEGIES FOR LOGIC PROGRAMS

D.A. Wolfram, M.J. Maher, J-L. Lassez
Department of Computer Science
University of Melbourne
Parkville, Victoria, 3052
Australia.

ABSTRACT

The treatment of soundness, weak completeness and strong completeness of various logic program resolution strategies with respect to success and failure is unified, generalized and considerably simplified. This is made possible by using the full power of the unification theorem which allows a reduction to a simple canonical case. The results can then be established in a natural and straightforward manner. We also indicate how the unification theorem can be used to simplify the proof of the completeness of the negation as failure rule. Finally we note that the treatment introduced in this paper applies to other clausal forms.

1 INTRODUCTION

SLD (or LUSH) resolution, on which most PROLOG interpreters are based, is SL-resolution (Kowalski and Kuehner 1971) restricted to Horn clause logic programs (Kowalski 1974). The soundness and completeness of SLD-resolution were first established in (Hill 1974). Further results, in particular on strong completeness can be found in (Clark 1979). (Apt and van Emden 1982) provided a more algebraic treatment in a first part of their important paper. In order to treat also the problem of negation as failure by algebraic means, Apt and van Emden in a second part of their paper provided a characterization of SLD finite failure. It was then shown in (Lassez and Maher 1982) that this characterization could be interpreted as a form of weak completeness of SLD-resolution with respect to finite failure, and they proceeded to give a strong completeness result for fair SLD-resolution. These results were compiled in (Lloyd 1982).

Unfortunately, the proofs of these results are long and involved, they rely on a proliferation of definitions and lemmas and many obscure points are left for the reader to verify.

Even though most, if not all, results are conceptually clear, their presentation is made complex by the presence of non-determinism. One of the crucial places where the existing proofs are very involved is where the order of selection of atoms is shown to be irrelevant; that is, different resolution strategies

This research is partially supported by the A.C.R.B.

lead to the same answer substitution. However it may be suspected that these equivalences are a direct consequence of an already existing powerful result embodying some kind of Church-Rosser property (or diamond lemma), in which case most of the proofs would be redundant and a major difficulty removed.

This powerful result is in fact Robinson's fundamental unification theorem (Robinson 1965). The formalism of (Martelli and Montanari 1982) in terms of a system of equations has been chosen, as it is more suitable for the purpose. A derivation (SLD or other) corresponds to solving equations step by step, leading eventually to a unification of the list of all atoms of the derivation with a list of corresponding heads of clauses. The different resolution strategies impose different orders in the selection of the equations to be solved. The unification theorem states that the ultimate result (a most general unifier or a failure) is independent of the order in which the equations are selected. This allows the non-deterministic aspects to be factored from the treatment of soundness and completeness, leaving a unique search space, the canonical tree, which corresponds to Breadth-First resolution (BF-resolution) and may serve as a high level basis to study and-parallelism.

The definitions of success and finite failure sets (van Emden and Kowalski 1976), (Lassez and Maher 1982) are given inductively and are found to correspond very clearly to ground versions of the inductive definitions for successful and failed BF-derivations respectively. The various proofs of soundness and completeness for BF-resolution become then essentially direct consequences of the definitions.

The analogous results for a number of resolution strategies are direct corollaries to those for BF-resolution by the previous treatment of non-determinism.

This point of view highlights the fundamental role played by the unification theorem, and leads to a more general, unified and straightforward presentation. Furthermore another aspect of the unification theorem allows us to simplify the proof of the completeness of the negation as failure rule.

The paper is organised in the following way : after this introduction there is a section containing the necessary notations, definitions and preliminary results. In the third section the equivalence between the canonical tree and those corresponding to other resolution strategies is established. The final sections contain the results of soundness and completeness for Breadth-First and other resolution strategies.

Note : This paper will form the basis of a chapter of the forthcoming book, *The Semantics of Logic Programs* (Lassez, Maher and Wolfram).

2 PRELIMINARIES

A few necessary standard definitions and results are briefly recalled in this section. The appropriate background for resolution can be found in (Chang and Lee 1973) and for logic programming in (Kowalski 1979).

2.1 The Syntax of Logic Programs

The sets of *variables*, *function symbols* and *predicate symbols* are disjoint sets.

A *term* is either a variable or a zero-place function symbol (*constant symbol*) or $f(t_1,\ldots,t_n)$, where f is an n-place function symbol and $t_1,\ldots,t_n$ are terms. It is assumed that there is a constant symbol in the set of function symbols.

An *atom* is either a zero-place predicate symbol (*proposition*) or $P(t_1,\ldots,t_n)$, where P is an n-place predicate symbol and $t_1,\ldots,t_n$ are terms.

The *principal function symbol* of $g(t_1,\ldots,t_n)$ is g, where g is an n-place function or predicate symbol and $t_1,\ldots,t_n$ are terms.

A *fact* is P_0, where P_0 is an atom.

A *rule* is

$$P_0 \leftarrow P_1,\ldots,P_n \quad (n > 0)$$

where $P_0,P_1,\ldots,P_n$ are atoms. The *head* of the rule is P_0 and the *body* of the rule is $P_1,\ldots,P_n$.

A fact or rule $P_0 \leftarrow P_1,\ldots,P_n$ $(n \geq 0)$, will sometimes be abbreviated to

$$H_j \leftarrow B_j.$$

where $H_j \equiv P_0$ and $B_j \equiv P_1,\ldots,P_n$ $(n \geq 0)$ and will be referred to as a *clause*.

A *logic program* is a finite set of clauses.

A *goal* is $\leftarrow P_1,\ldots,P_n$, where $P_1,\ldots,P_n$ are atoms. A goal is the *empty goal* (□) if n is zero.

2.2 Substitutions and Unification

2.2.1 Substitutions

A *substitution* θ is a finite set of *components* :

$$\theta = \{t_1/x_1,\ldots,t_n/x_n\}$$

where $t_1,\ldots,t_n$ are terms and $x_1,\ldots,x_n$ are distinct variables.

The substitution θ is a *ground substitution* if no variables occur in $t_1,\ldots,t_n$. The substitution θ is a *renaming* substitution if $t_1,\ldots,t_n$ are distinct variables and $t_i \neq x_i$ $(i = 1,\ldots,n)$. The *empty substitution* is $\varepsilon = \{\}$.

The *instance* of a finite string of symbols E by θ, denoted by $E\theta$, is obtained by simultaneously replacing each occurrence of x_i $(i = 1,\ldots,n)$ in E by t_i. A *ground instance* of E is an instance in which no variable occurs.

The *composition* of two substitutions $\theta = \{t_1/x_1,\ldots,t_n/x_n\}$ and λ, denoted by $\theta\lambda$, is $\theta' \cup \lambda'$ where θ' is the set of all components $t_i\lambda/x_i$ $(i = 1,\ldots,n)$, such that $t_i\lambda$ differs from x_i, and λ' is the set of all components s/y of λ, where y is not among $x_1,\ldots,x_n$. It can be shown that $(\theta\lambda)\delta = \theta(\lambda\delta)$, and $(E\theta)\lambda = E(\theta\lambda)$, for any substitutions θ, λ and δ.

2.2.2 Unification

Two finite strings of symbols E and F are *unifiable* if there is a substitution θ, called a *unifier* of E and F, such that $E\theta$ is identical to $F\theta$.

The unifier μ is a *most general unifier* (*mgu*) of E and F if and only if for every unifier θ of E and F there is a substitution β such that $\theta \equiv \mu\beta$.

2.2.3 Non-Deterministic Unification

The following is a statement of one of Martelli and Montanari's versions (Martelli and Montanari 1982) of the fundamental unification algorithm and theorem in (Robinson 1965). (The algorithm has been slightly reworded so that it applies to the unification of atoms.)

Unification Algorithm

Given a set of equations $X = \{t_i = t'_i \mid i = 1, \ldots, k\}$ where t_i and t'_i are atoms, repeatedly perform any of the following transformations. If no transformation applies, stop with success.

(a) Select any equation of the form $t = x$ where t is not a variable and x is a variable, and rewrite it as $x = t$.

(b) Select any equation of the form $x = x$ where x is a variable, and erase it.

(c) Select any equation of the form $t' = t''$ where t' and t'' are not variables. If the two principal function symbols are different, stop with failure. If the two principal function symbols are constants, erase the equation. Otherwise, $t' = t''$ is of the form $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ and replace $t' = t''$ by: $t_1 = t'_1, \ldots, t_n = t'_n$.

(d) Select any equation of the form $x = t$ where x is a variable which occurs somewhere else in the set of equations and where $t \neq x$. If x occurs in t, then stop with failure; otherwise apply the substitution $\sigma = \{t/x\}$ to all other equations (without erasing $x = t$).

Unification Theorem

(i) The unification algorithm terminates no matter which choices are made.

(ii) If the unification algorithm terminates with failure, X has no unifier. If it terminates with success then

(1) the equations are in the form $x_j = t_j$, $j = 1, \ldots, n$ where x_j is a variable and t_j is a term.

(2) every variable which is on the left side of an equation occurs only there.

(3) an mgu μ for X is $\{t_1/x_1, t_2/x_2, \ldots, t_n/x_n\}$

3 RESOLUTION STRATEGIES

In SLD-resolution, at each resolution step a single atom is selected from the current goal, to be unified with a head of a clause. In Breadth-First resolution (BF-resolution) the whole list of atoms of the goal is unified with a whole list of corresponding heads. GLD-resolution (Generalised Linear resolution for Definite clauses) covers these two extreme cases and the intermediate cases : at each step a non-empty subset of the set of atoms of the current goal is selected for unification.

Hence, depending on the choice of unification strategy it may not be possible to simulate strictly GLD-resolution by SLD-resolution. GLD-resolution may serve as a basis for the study of (partial) and-parallelism. It therefore differs from SLD-resolution in two respects : one is the selection of atoms to be unified, and the other is the unification itself.

As with SLD-resolution, different resolution strategies lead to different search spaces.

however, BF-resolution has a uniquely defined search space : the canonical tree.

This natural definition allows the proofs of the equivalences between all GLD-resolution strategies and the BF one to be simply formulated.

The following definitions give the necessary precisions.

3.1 GLD-derivations and GLD-trees

3.1.1 GLD-derivations

As a straightforward generalization of SLD-resolution, the aim of a GLD-derivation is to find a substitution μ, called the *answer substitution.*

A *GLD-derivation* for $P \cup \{G_0\}$, where P is a logic program and G_0 is a goal, is defined as follows :

Let
$G_l \equiv \leftarrow A_1,\ldots,A_n$ $(l \geq 0,\ n \geq 0)$

If $n = 0$, the GLD-derivation is a *success of length l* and the *answer substitution* μ is $\mu_0\mu_1\mu_2\ldots\mu_{l-1}$.

If $n > 0$, and there is an *input list* I_l of m_l clauses $H_j \leftarrow B_j$ $(1 \leq j \leq m_l \leq n)$, which

(a) are any m_l clauses of P to which renaming substitutions have been applied so that a variable in one of them does not occur either in the others, G_k or I_i $(0 \leq k \leq l,\ 0 \leq i < l)$, and

(b) $(H_1,\ldots,H_{m_l})$ and a list $(C_1,\ldots,C_{m_l})$ of m_l *selected atoms* from G_l are unifiable with mgu μ_l

Then, G_{l+1} is the goal obtained by applying μ_l to G_l and replacing $C_j\mu_l$ by $B_j\mu_l$ $(1 \leq j \leq m_l)$.

Otherwise, the GLD-derivation is a *failure of length l.*

Remark. By the choice of renaming substitution in (a), $\mu_0, \mu_1, \ldots, \mu_{l-1}$ do not affect H_j or B_j, so that $H_j\mu_l \equiv H_j\mu_0\mu_1\ldots\mu_l$ and $B_j\mu_l \equiv B_j\mu_0\mu_1\ldots\mu_l$. Therefore, atoms of G_{l+1} can be written $X\mu_0\mu_1\ldots\mu_l$ where X is the original atom, appearing either in G_0 or in the body of some clause used for replacement. This fact will be used later.

A GLD-derivation is a *BF-derivation* when $m_l = n$. A GLD-derivation is an *SLD-derivation* when $m_l = 1$.

A *fair GLD-derivation* is either a failed GLD-derivation or one in which an instantiated copy of every atom in a goal is a selected atom after a finite number of derivation steps. By definition, every BF-derivation is fair.

3.1.2 GLD-trees

As in the case of SLD-resolution, a GLD-tree represents a search space for a successful derivation.

For a given strategy of selecting atoms, the *GLD-tree* for $P \cup \{G_0\}$, where P is a logic program and G_0 is a goal, is defined as follows :

(1) G_0 is the root of the tree.

(2) The descendants of G_l are the goals which can be GLD-derived in one step.

A *successful branch* or *failed branch* of a GLD-tree

For any given atom in a goal it is easy to find its introduced version in the derived goals until eventually it is selected in some goal G_k and replaced by a body and a corresponding equation which appears at the end of G_{k+1}. By the representation, there is no ambiguity about the association of selected atoms, the bodies that are used to replace them and the equations.

To show that two resolution strategies are equivalent it is only necessary to make sure that they select the same atoms and replace them using the same clauses. The following algorithm and theorem are used in establishing equivalences between GLD-trees and therefore GLD-resolution strategies.

Let T_1 and T_2 be GLD-trees for $P \cup \{G_0\}$. The following algorithm reconstructs a derivation $\{G_i\}$ in T_2 from a given fair non-failed derivation in T_1.

Reconstruction Algorithm

Step i, i = 0,1,...

For every selected atom A in G_i of T_2 :

If A appears in G_0 then trace the corresponding atom A of T_1 down the given derivation until it is selected and replaced by a body B and an equation A = H.

Otherwise A is introduced in some G_k as part of the body replacing some atom C. Find the corresponding C in T_1 (this must have been done to perform the replacement) and trace down the given derivation to where A is selected and replaced by a body B and an equation A = H.

Perform the replacement of A by B in G_i and add A = H to the set of equations at the end of G_i.

Once this is done, G_{i+1} has been formed.

The tracing of an atom down the derivation in T_1 always terminates since the derivation is fair.

If the given derivation is successful (and so finite) then the derivation which is constructed must also be finite. It is easy to verify that the derivation in T_1 and the reconstructed derivation in T_2 define the same set of equations and therefore, by the unification theorem, the reconstructed derivation is successful and both derivations lead to the same answer substitution (mgu of the equations).

If the given derivation is (fair) infinite then the constructed derivation is also infinite.

This gives :

Theorem 3.1

Let T_1 and T_2 be GLD-trees for $P \cup \{G_0\}$.

(1) If T_1 has a successful branch then so does T_2.

(2) If T_1 has an infinite fair branch then T_2 has an infinite branch.

From this theorem and the preceding discussions, the following equivalences can be immediately deduced. These equivalences will allow the treatment of soundness and completeness (for success and finite failure) for GLD-resolution to be reduced to BF-resolution and its

associated canonical tree.

Corollary 3.2

The following statements are equivalent :

(1) The canonical tree has a successful branch with answer substitution μ.

(2) There is a GLD-tree with a successful branch with answer substitution μ.

(3) Every GLD-tree has a successful branch with answer substitution μ.

The equivalence of fair GLD-trees with respect to finite failure is shown by the following theorem, which is a consequence of Theorem 3.1.

Theorem 3.3

Let T_1 and T_2 be GLD-trees for $P \cup \{G_0\}$. If T_1 is finitely failed and T_2 is fair, then T_2 is finitely failed.

Proof

Suppose that T_2 has a successful or (fair) infinite branch. As T_2 is fair, T_1 must have a successful or infinite branch by theorem 3.1. This is a contradiction. Hence, T_2 is finitely failed. □

Corollary 3.4

The following statements are equivalent :

(1) The canonical tree is finitely failed.

(2) There is a finitely failed GLD-tree.

(3) Every fair GLD-tree is finitely failed.

4 SOUNDNESS AND COMPLETENESS

4.1 Success and Finite Failure

Sets

There are a number of ways of giving the semantics of logic programs : least model, least fixedpoint (van Emden and Kowalski 1976), denotational (Lassez and Maher 1983), optimal fixedpoint (Lassez and Maher 1983), tree rewriting system (Colmerauer 1982), etc. The definition chosen here formalizes the intuitive notion that a logic program P, viewed as a production system, defines inductively a set of true ground facts called the *success set*. These facts are either given in the program or are derived by repeatedly applying the rules.

Let

$$S = \bigcup_{i \geq 0} S_i, \text{ where}$$

$S_0 = \emptyset$, and

$A \in S_i$ if and only if $B_0 \leftarrow B_1, \ldots, B_n$ $(n \geq 0)$ is a ground instance of a clause of P such that $A \equiv B_0$ and $\{B_1, \ldots, B_n\} \subseteq S_{i-1}$.

It is straightforward to verify that the success set is identical to the least model of P (van Emden and Kowalski 1976), and therefore $P \cup \{G_0\}$ is inconsistent if and only if the atoms of a ground instance of the goal G_0 are in S (Apt and van Emden 1982).

An inductive definition is also given of a set of ground facts which can effectively be shown not to belong to the success set. This set is called the *finite failure set*. It was introduced in (Lassez and Maher

1982) to establish the soundness and strong completeness for finite failure of fair SLD-resolution, and used in (Jaffar, Lassez and Lloyd 1983) to establish a completeness result for the negation as failure rule (Clark 1978). The definition chosen here is slightly different in form from the original definition, as it helps to unify the treatment of success and failure.

Let

$$FF = \bigcup_{i \geq 0} FF_i, \text{ where}$$

$A \in FF_i$ if and only if $A \notin R_i$.

$B \in R_0$ if and only if $C_0 \leftarrow C_1,\ldots,C_n$ $(n \geq 0)$ is a ground instance of a clause of P such that $B \equiv C_0$

$B \in R_i$ if and only if $C_0 \leftarrow C_1,\ldots,C_n$ $(n \geq 0)$ is a ground instance of a clause of P such that $B \equiv C_0$ and $\{C_1,\ldots,C_n\} \subseteq R_{i-1}$.

4.2 Soundness and Completeness of BF-resolution

Throughout the remainder, P represents a logic program and G_0 a goal.

The following lemma is a major tool used in proving the completeness of BF-resolution for success, and its soundness for finite failure. As the production of an element of S_i or R_i is tantamount to a ground BF-derivation, by "lifting" such a derivation to the form of a BF-derivation, the results can be established directly. In the lemma, Y stands for either S or R.

Lifting Lemma

If G_0 is the goal $\leftarrow A_1,\ldots,A_n$ and there is a substitution α_0, such that the atoms of $G_0\alpha_0$ are in Y_l, then there is a BF-derivation step from G_0 to G_1. Furthermore, when $(l > 0)$ there is a substitution α_1 such that the atoms of $G_1\alpha_1$ are in Y_{l-1} and $\alpha_0\rho_0 \equiv \mu_0\alpha_1$, for some substitution ρ_0.

Proof

By definition of Y, there are n ground instances of clauses of P, $H_j\gamma_j \leftarrow B_j\gamma_j$ $(j = 1,\ldots,n)$, where $H_j \leftarrow B_j$ is a clause of P to which a renaming substitution has been applied and γ_j is a ground substitution, such that $A_j\alpha_0 \equiv H_j\gamma_j$ and when $l > 0$ the atoms of $B_j\gamma_j$ are in Y_{l-1}.

Let $\rho_0 = \gamma_1\ldots\gamma_n$ and I be the list $\{H_j \leftarrow B_j \mid j = 1,\ldots,n\}$. It can be assumed that a variable in a clause of I_0 does not occur in G_0 nor in another clause of I.

Hence, $(H_1,\ldots,H_n)\alpha_0\rho_0 \equiv (A_1,\ldots,A_n)\alpha_0\rho_0$, so there is an mgu μ_0 and a substitution α_1 such that $\alpha_0\rho_0 \equiv \mu_0\alpha_1$.

By the definition of BF-derivation, there is a derivation step from G_0 to G_1 with input list $I_0 = I$ and mgu μ_0. Furthermore, when $l > 0$ the atoms of $G_1\alpha_1 \equiv \leftarrow B_1\mu_0\alpha_1,\ldots, B_n\mu_0\alpha_1$ are in Y_{l-1}. □

4.2.1 Soundness and Completeness for Success

Let the atoms of a goal G be in S_i. These atoms are produced from elements in S_{i-1} which are

in turn produced from elements in S_{i-2} and so on. From the definition of S, it is easy to build what is tantamount to a ground BF-derivation for P U {G}.

The problem of soundness therefore becomes simple : if ground instances of the goals of a BF-derivation are formed, the atoms in the initial goal belong to S by the direct application of the definition.

From corollary 3.2, the answer substitution μ is a most general solution of the equations associated to a BF-derivation. The following theorem establishes that all of the atoms of any ground instance of $G_0\mu$ are in the success set.

Theorem 4.1

If P U $\{G_0\}$ has a successful BF-derivation of length l, then the atoms of any ground instance $G_0\mu\alpha$ of G_0 are in S_l.

Proof

Apply $\mu\alpha$ to all the goals of the derivation and apply any substitution so that no variable remains.

By the definitions of BF-derivation and S, if all atoms of the ground instance of the goal G_n are in S_{i-1}, then all atoms of the ground instance of G_{n-1} are in S_i.

As G_l is the empty goal, its atoms are in $\emptyset = S_0$.

The proposition follows. □

The following completeness theorem is a further illustration that the answer substitution μ is a most general solution.

Theorem 4.2

If there is a substitution α_0 such that the atoms of $G_0\alpha_0$ are in S_l $(l > 0)$, then P U $\{G_0\}$ has a successful BF-derivation of length l, such that $G_0\alpha_0 \equiv G_0\mu\alpha_l$, for a substitution α_l.

Proof

The atoms of $G_0\alpha_0$ are in S_l, so by repeated applications of the lifting lemma, there are l BF-derivation steps from G_0 to G_l and substitutions α_k such that the atoms of $G_k\alpha_k$ are in S_{l-k} $(0 \leq k \leq l)$.

Furthermore,
$\alpha_i\rho_i \equiv \mu_i\alpha_{i+1}$ $(0 \leq i < l)$.

Hence,

$$\alpha_i\rho_i\rho_{i+1} \equiv \mu_i\alpha_{i+1}\rho_{i+1} \equiv \mu_i\mu_{i+1}\alpha_{i+2}$$

and

$$G_0\alpha_0 \equiv G_0\alpha_0\rho_0\ldots\rho_{l-1} \equiv G_0\mu_0\ldots\mu_{l-1}\alpha_l$$

$$\equiv G_0\mu\alpha_l. \quad \square$$

4.2.2 Soundness and Completeness for Finite Failure

The following theorem establishes the soundness of BF-resolution for finite failure.

Theorem 4.3

If every BF-derivation for P U $\{G_0\}$ is failed by length $< l$, then every ground instance of G_0 contains an atom in FF_l.

Proof

The proof is by induction on l.

If every BF-derivation for P U $\{G_0\}$ is failed by length zero, then every ground instance of G_0 contains an atom in FF_0. Otherwise, by the lifting lemma, there would be a BF-derivation

step from G_0.

The induction hypothesis is that the theorem is true for l-1.

If every BF-derivation for G_0 is failed by length $< l$ then, by the induction hypothesis, every ground instance of every G_1 contains an atom in FF_{l-1}. Suppose there is a substitution α_0 such that the atoms of $G_0\alpha_0$ are in R_l. Then by the lifting lemma, there is a BF-derivation step from G_0 to G_1 and a substitution α_1 such that the atoms of $G_1\alpha_1$ are in R_{l-1}. This is a contradiction.

Therefore every ground instance of G_0 contains an atom in FF_l. □

The next theorem establishes the completeness of BF-resolution for finite failure.

Theorem 4.4

If every ground instance of G_0 contains an atom in FF_l, then every BF-derivation for $P \cup \{G_0\}$ is failed by length $< l$.

Proof

The proof is by induction on l.

If every ground instance of G_0 contains an atom in FF_0, then every BF-derivation for $P \cup \{G_0\}$ is failed by length zero. Otherwise, by the definitions of BF-derivation and R_0, a ground instance of G_0 could be found which would not contain an atom in FF_0.

The induction hypothesis is that the theorem is true for l-1.

Let every ground instance of G_0 contain an atom in FF_l. If G_0 has no descendants, then it is failed by length $< l$. Otherwise, if every ground instance of every descendant G_1 contains an atom in FF_{l-1} then, by the induction hypothesis, every G_1 is failed by length $< l-1$ and therefore G_0 is failed by length $< l$. If neither of these cases hold, then there is a substitution β such that the atoms of $G_1\beta$ are in R_{l-1}. By the definitions of BF-derivation and R_l, the atoms of a ground instance of $G_0\mu_0\beta$ are in R_l. This is a contradiction.

Hence, every BF-derivation for $P \cup \{G_0\}$ is failed by length $< l$. □

5 CONCLUSION

The soundness and completeness of BF-resolution and the equivalences of GLD-trees are used here to prove soundness and completeness results for GLD-resolution.

Theorem 5.1

The following statements are equivalent.

(1) There is a substitution α_0 such that the atoms of $G_0\alpha_0$ are in S_l for some l.

(2) $P \cup \{G_0\}$ is inconsistent.

(3) $P \cup \{G_0\}$ has a successful GLD-derivation with answer substitution μ, and $G_0\alpha_0 \equiv G_0\mu\alpha_l$.

(4) The canonical tree has a successful branch of length l with answer substitution μ, and $G_0\alpha_0 \equiv G_0\mu\alpha_l$.

(5) There is a GLD-tree with a successful branch with answer substitution μ, and $G_0\alpha_0 \equiv G_0\mu\alpha_1$.

(6) Every GLD-tree has a successful branch with answer substitution μ, and $G_0\alpha_0 \equiv G_0\mu\alpha_1$.

Theorem 5.2

The following statements are equivalent for $P \cup \{G_0\}$.

(1) Every ground instance of G_0 contains an atom in FF.

(2) Every fair GLD-derivation is finitely failed.

(3) The canonical tree is finitely failed.

(4) There is a finitely failed GLD-tree.

(5) Every fair GLD-tree is finitely failed.

In (Lassez, Maher and Wolfram) similar techniques are used to simplify the treatment of the soundness and completeness of the negation as failure rule. In particular the equivalence relation $\downarrow$ used extensively in (Jaffar, Lassez and Lloyd 1983), can be replaced by the following relation :

$$s \star t \text{ iff } \exists n : s\theta_0..\theta_n \equiv t\theta_0..\theta_n$$

That $\star$ is an equivalence relation follows directly from property (2) of the Unification Theorem. The domain $D = T/\downarrow$ is replaced by the domain $D = T/\star$ and the required axioms (Clark 1978) are satisfied trivially. There is therefore no need to consider the involved formalisms of tree rewriting systems.

Consequently the Unification Theorem plays a central role in all basic results of soundness and completeness in the theory of logic programs.

The technique introduced in this paper is not merely restricted to GLD-resolution. As illustrated in the example it can be extended to other resolution strategies such as bottom-up and the intermediate ones.

It is a universal technique in resolution because it also provides a method to prove equivalences between resolution strategies for arbitrary clauses. The structure of the canonical strategy would depend on the form of the clauses involved.

ACKNOWLEDGEMENTS

We would like to thank Lee Naish and Rodney Topor for their corrections and helpful criticisms of a first draft of this paper.

REFERENCES

Apt, K.R. and van Emden, M.H. Contributions to the theory of logic programming. J ACM **29**,3 841-862, 1982.

Chang, C.L. and Lee, R.C.T. *Symbolic Logic and Mechanical Theorem-Proving*. Academic Press, New York, 1973.

Clark, K.L. Negation as Failure. in : *Logic and Data Bases*. H. Gallaire and J. Minker Eds., Plenum Press, New York, 293-324, 1978.

Clark, K.L. Predicate logic as a computational formalism. Research Report 79/59, Dept. of Computing, Imperial College, London, 1979.

Colmerauer, A. Prolog II Manuel de référence et modèle théorique. rapport de recherche GIA ERA CNRS 363, Université d'Aix-Marseille II, Aix-en-Provence, 1982.

Hill, R. LUSH-resolution and its completeness. DCL Memo. 78, Dept. of Computational Logic, University of Edinburgh, 1974.

Jaffar, J., Lassez, J-L. and Lloyd, J.W. Completeness of the negation as failure rule. Proc. Eighth IJCAI, Karlsruhe, 500-506, 1983.

Kowalski, R.A. Predicate logic as a programming language. *Information Processing 74*, J. Rosenfeld Ed., North-Holland, Amsterdam, 556-574, 1974.

Kowalski, R.A. *Logic for Problem-Solving*, North-Holland, New York, 1979.

Kowalski, R and Kuehner, D. Linear resolution with selection function. Artificial Intelligence **2** 227-260, 1971.

Lassez, J-L. and Maher, M.J. Closures and fairness in the semantics of programming logic. Theor. Comp. Sci. (to appear). Revised version of TR/6, 1982.

Lassez, J-L. and Maher, M.J. The denotational semantics of Horn clauses as a production system. Proc. of the National Conference on Artificial Intelligence, AAAI-83, Washington D.C., 229-231, 1983.

Lassez, J-L. and Maher, M.J. Optimal fixedpoints of logic programs. Third International Conference on the Foundations of Software Technology and Theoretical Computer Science, Bangalore, 343-362, 1983.

Lassez, J-L., Maher, M.J. and Wolfram, D.A. *The Semantics of Logic Programs*. Oxford University Press, (in preparation).

Lloyd, J.W. Foundations of logic programming. Technical Report, Dept. of Computer Science, University of Melbourne, TR/7 1982. (revised 1983)

Martelli, A. and Montanari, U. An efficient unification algorithm. ACM TOPLAS **4**,2 258-282, 1982.

Robinson, J.A. A machine-oriented logic based on the resolution principle. J ACM **12**,1 23-41, 1965.

van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language. J ACM **24**,4 733-742, 1976.

FAME: A PROLOG PROGRAM THAT SOLVES PROBLEMS IN COMBINATORICS

Yoav Shoham
Computer Science Department
Yale University
P.O.Box 2158 Yale Station
New Haven, CT 06520, USA

[1] **Abstract:** FAME is a Prolog program that solves problems in combinatorics. The nature of the task and solution methodolgy are discussed. The program and the algorithms involved are described in some detail. A special emphasis is put on the choice of Prolog as an implementation language.

1 Overview: task and methodology

This is a report on FAME, a Prolog program that solves problems in combinatorics. Combinatorics is a difficult domain for students to solve problems in. What are the insights and inspirations that problems in combinatorics seem to require, and that frustrate the student who was doing just fine on integration problems? Whatever the correct answer may be, one hopes that it will shed light on the nature of intelligence.

The ultimate goal is for the program to have problem-solving capabilities similar to that of student who has had one course in discrete mathematics. For example, I require that the program should solve the following problem set given to an undergraduate theory course:

1. Give a combinatorial argument for the following equalities: C(N-1,R) = (R+1)C(N,R+1) = (N-R)C(N,R).
2. Explain why the number of ways to put N indistinct objects into K distinct boxes is C(N+K+1,K-1).
3. How many ways are there to put N indistinct objects into K distinct boxes where every box receives at least one of the objects? How does this problem relate to the one before?
4. The following problem is related to problem 3. In an arrangement of 11 consecutive seats, how many ways are there to select 4 seats so that no two are adjacent? Explain your answer of course.
5. Give a combinatorial argument that SIGMA(I

[1]This work was supported in part by the Advanced Research Projects Agency of the Department of Defence and monitored under the Office of Naval Research under contract N00014-83-K-0281.

```
from 0 to N,C(N,I)**2)
= C(2N,N).
```

(there were three more questions, but these five provide more than enough material for thought).

The problems to be solved by the program are counting problems or closely related ones. Counting problems have the form "In how many ways can you.." or "How many X are there such that Y". Partition problems are a special case of counting problems, and have the form "In how many ways can you partition X into Y such that...". I am not interested in a program that can deal solely, say, with proof of binomial equalities or solely with partition problems. This demanding criterion has an effect that the program must largely imitate human problem solving in the domain. Solutions to special classes of problems that are counterintuitive tend not to extend well to the rest of the problems in the domain. On the other hand I do not demand that the program be complete for any class of problems it solves. As Boyer and Moore put it [2],

> It has been argued that mechanical theorem-proving is an impossible task because certain theories are known to be undecidable or super-super-exponential in complexity. Such metamathematical results are, of course, no more of an impediment to mechanical theorem-proving than to human theorem-proving. They only make the task more interesting (p.6)

Consider for example the class of problems involving proof of combinatorial equalities, of which the first of the above problems is an instance. The excellent result of Zeilberger [16] includes a procedure for solving a very wide class of those problems, much wider than any student could solve. Yet that procedure is not extendable to counting problems in general and partition problems in particular. Indeed, Zeilberger's procedure bears little resemblance to the method employed by the average student. Similar remarks apply to Gosper's work [4].

In contrast to that, the part of FAME which solves combinatorial equalities closely resembles human problem-solving in that domain. The general structure of proving an equality by a combinatorial argument is to demonstrate that both are a correct answer to the same counting problem. One typically constructs a problem (which I call a *story*) by analyzing one expression. A story describes what is to be counted. Typically it contains a list of sets and their cardinalities, and a list of relations that hold between the sets.[2] The second step is to show that the other expression is also a solution to that same problem, if the counting is done differently. Since a story has a unique solution, this is a valid proof.

For example, the first problem from the above problem set reads:
Give a combinatorial argument for the following equalities:
N*C(N-1,R) = (R+1)*C(N,R+1) = (N-R)*C(N,R)
To prove these equalities think in how many ways you could choose a team of R+1 players and appoint a captain from among the players, from a given class of N people. The different expressions correspond to whether you first choose the captain, the rest of the team, or the whole team. The part of FAME which performs this kind of reasoning is described in section 2.

One should note that the task is not a purely mathematical one. Consider the five questions presented above. The problems 1 and 5 require

---

[2]Actually things get more complex, and later on stories will contain existential and universal quanifiers, formal sets and other creatures.

a combinatorial proof. Problems 2 and 3 speak about putting objects into boxes, and more importantly state that the objects or boxes are (in)distinct. Problem 4 speaks about adjacency, and even hints at a relation to distinctness. How are all these underlined concepts to be represented so as to facilitate effective reasoning, and hopefully reflect human understanding of those concepts? The mathematician could no doubt provide helpful insight into mathematics (just as a doctor could help in building a medical diagnosis expert system), but the task as a whole is a metamathematical one.

The view of mathematical problem solving here is as a sort of planning activity, in the sense widely used in AI ([11],[13],[9]). We formulate strategies for solving a problem, trying them out according to certain rules, constantly monitoring our progress - deciding on resource allocations and debugging solutions. However, for any planning to take place we need "planning material", a structured domain. In an impoverished domain there is no need for planning, and in a large but structureless domain planning is not possible. So my main concern is to conceptualize a framework in which the problems can be solved, and then to verify its validity by solving them.

The criteria imposed on the program are that the representation of the input and output correspond to their representation in the "real world", and that the program should be robust. For example, in solving the first problem from the homework assignment I demanded that the solution should hold for all "similar" problems (for a more detailed explanation see the section 2).

If the task is not a purely mathematical one, the work described here is also different from work done by psychologists on mathematical problem-solving, for example the work of Kant and Newell [5] and that of Anderson et al. [1], in that the main goal is performance and robustness of the system rather than a psychologically valid imitation of human problem-solving behavior. Closer in spirit is McAllester's work [8], though there the stress on robustness and depth of understanding is carried to an extreme, and a general representation scheme of mathematical objects is devised. The intention there, as I understand it, is to write robust and natural problem solvers in the future using that representation.
My approach is more experimental. By trying to actually solve the problems I am forced to deal with the problems as they arise. Some of my solutions take a form of a representation that seems to fit in well with McAllester's symmetric set theory. Others do not fit in so neatly; in particular, I rely strongly on the notion of a natural number to represent set cardinalities. The overall effect is that the spirit of FAME lies somewhere between those of McAllester's work on the one hand, and MACSYMA [7] and SMP [15] on the other. It should be noted that neither MACSYMA nor SMP can be applied to the problems which motivated the construction of FAME (and in particular to the sample problem set described in the first section).

FAME actually has two versions, called FAME I and FAME II. Lessons learned in constructing FAME I have been applied to FAME II, which is undergoing daily changes. Due to its evolving state and wide scope FAME II will only be sketched out here, and a full description of it will be given elsewhere. Both programs are written in PROLOG.[3] The choice was first made on the basis of a fairly obscure personal liking of the language. The work on FAME turned this vague liking to an accountable commitment, and it will be shown why this was so. The value of Prolog was not always

[3]The exact environment is a DEC-20 running the Warren/Pereira/Byrd Prolog-10, version 3

its non-deterministic and declarative nature, as defined in theoretical work such as [14] or fully applied in [12]. In the parts where it is used declaratively it is most elegant, and this will be demonstrated in the paper. For other parts of the program the natural interpretation is the procedural one, and the combination of declarative islands in a procedural stream is a very convenient paradigm. The factors which made Prolog such a fortunate choice of implementation language will be summarized after FAME I is discussed.

2 FAME I

2.1 General description

The first class of problems to be attacked was proof of combinatorial equalities. The input is a pair of expressions, and the output is a proof of equality. FAME I only deals with expressions that are integers (1,2,..), symbolic (n,m,..), have the form C(X,Y) (where C(X,Y) denotes "X choose Y") or (recursively) have the form E1*E2. The X and Y in the previous sentence may be any expressions involving integers, symbols and the operators + and -. For example, the expressions could be 13, k, c(n,k), c(n-1,r+2)*c(r-1,4), but not c(n*k,r) or c(n,k)+c(n,n-k).

The first obvious observation is that by definition the "combinatorial argument" requirement precludes the solution that expands C(x,y) into the x!/(y!*(x-y)!), and proceeds with number-theoretic arguments. As David McAllester put it[4], C(x,y) can be viewed as one of at least two different things: the algebraic expression x!/(y!*(x-y)!) or the number of different y-sized subsets of a given x-sized set for fixed x and y. In combinatorial arguments we require the latter interpretation. The reason I am not interested in the former solution is not only because of the phrasing of the question, but because that kind of solution will not extend to counting problems. Also, it is this level of reasoning that facilitates the clever tricks and insights, and on which a two-line solution can be given to problem 5 from the previous section.

The previous section outlines the human method for "combinatorially" proving combinatorial equalities, which is how I started to construct FAME I. The first step was to have an algorithm for creating stories from expressions. At this point I slightly varied the method described above, realizing that the task of matching a given story and a given expression is in part very similar to the story creation task. Having already coded up the story creation, I asked what would happen if I created independent stories for the two expressions - would that be useful? The answer turned out to be yes; all you have to do is transform the two stories to a certain canonical form, and compare the canonical forms. Those must be the same up to isomorphism, which is what the program checks for. The exact procedure is described in detail in the next section.

2.2 Algorithm and implementation

The algorithm for testing whether two such expressions are equal is as follows.

Algorithm 1: Proving a combinatorial equality

1. 1. Create a story Story1 for Exp1, and find its canonical form CanStory1.
2. 2. Create a story Story2 for Exp2, and find its canonical form CanStory2.
3. 3. Prove that CanStory1 and CanStory2 are

[4] personal communication

isomorphic.

The story creation algorithm is non-deterministic and rule-based. Its input is an expression, and its output is a story which is a tuple *<Sets,Set-Relations>*. *Sets* is a set of tuples *<SetName,Cardinality>*. *Set-Relations* is a list of predicates denoting set relations, each of which has the form *subset(SetName1,SetName2)* or *partition(List-of-SetNames,SetName)*. The algorithm picks the terms of the input expression one at a time (in a non-deterministic order), generates a new set name (or more than one, if needed) and augments the story by adding to it the new sets and appropriate set relations.

Example 1
Suppose the input expression is n*C(n-1,r). One behavior of the program could be as follows. The expression n is known to be symbolically equal to C(n,1). The sets *set1* of cardinality 1 and *set2* of cardinality n are created, and the story at this point is
`story([(set1,1),(set2,n)],[subset(set1,set2)])`. Which translated into English, reads "In how many ways can you choose a set of size 1 from a set of size n?".
The next and last term is C(n-1,r). The program creates two more set names, *set3* associated with a cardinality of n-1 and *set4* associated with a cardinality of r. It also notices that the cardinalities of set3 and set1 sum up to the cardinality of set2, so the final story is

```
story(
 [(set1,1),(set2,n),
  (set3,n-1),(set4,r)],
 [partition([set1,set3],set2),
  subset(set4,set3)]),
```

or in English: "In how many ways can you partition a set of size n into two sets of sizes 1 and n-1, and choose a set of size r from the latter?".

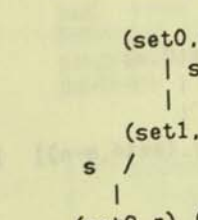
Things get more interesting in the remainder of the algorithm. The canonicalization algorithm is as given below. The input to the algorithm is a story, and the output is the same story in its canonical form.

Algorithm 2: Deriving a canonical form of a story
The algorithm has three steps.

1. **Split**
2. **Pad**
3. **Flatten**

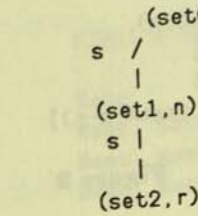
Instead of writing the details of the algorithm, it will be explained via the following example. Note: the **Flatten** phase is essentially identical to the algorithm used in the example on page 21 of [2], coincidentally also called **Flatten**.

Example 2 Let the input to the canonicalization algorithm be the story " In how many way can you choose n people from a total of m candidates, and from those n people construct a baseball team of r players and a football team of k players? A player may participate in both teams." Graphically, the story is described by



(The notation is (Setname,Cardinality). s=subset, p=partition).

After one **Split** the story is



After the second and last split the story is

```
(set0,m) (set01,m)
 s |        | s
   |        |
(set1,n) (set11,n)
 s |        | s
   |        |
(set2,r) (set3,k)
```

Pad changes all *subsets* to *partitions*, and the result is is shown in Figure 1. The final step **Flatten** yields the canonical form which is shown in Figure 2. The English form of the final story is `In how many ways can you partition a set of size m into three sets of sizes r, n-r and m-n, and another set of size m into three sets of sizes k, n-k and m-n?`

Already here one can see how Prolog is elegant and concise. For example, the following code for **Pad** is a direct encoding of the algorithm *If there exists a subset relation in the story then replace it by a partition and repeat, else return the story*:

```
pad(Sets,SR,NewSets,NewSR) :-
 remove(
  subset(Set,Set1),SR,TmpSR),!,
 gensym(set,Set2),
 member((Set,X),Sets),
 member((Set1,Y),Sets),
 pad([(Set2,Y-X)|Sets],
     [part([Set,Set2],Set1)
                  |TmpSR],
     NewSets,NewSR).
pad(Sets,SR,Sets,SR).
```

The last step in testing the equality is to test for story isomorphism. Since the canonical stories contain only one sort of relation (partition), and the only other information is the set cardinalities, the canonical story can be viewed as a forest of directed ("rooted") trees with labelled nodes:[5] the nodes are the sets, the labels are

```
                (set0,m)                    (set01,m)
                 p |                          | p
                   |                          |
          [(set1,n),(set4,m-n)]      [(set11,n),(set5,m-n)]
            p |                            | p
              |                            |
[(set2,r),(set6,n-r)]          [(set3,k),(set7,n-k)]
```

Figure 1

```
                (set0,m)                    (set01,m)
                 p |                          | p
                   |                          |
[(set2,r),(set6,n-r),(set4,m-n)]   [(set3,k),(set7,n-k),(set5,m-n)]
```

Figure 2

the cardinalities, and the set of a node's "sons" is a partition of the set denoted by the node. Before the explicit algorithm for testing the isomorphism is given, the reader's attention is first drawn to the following logical equivalence:

Two dl-forests T(V,E) and S(U,F) are isomorphic
iff
there exists a pairing $P =$
$\{<v,u>: v \text{ in } V, u \text{ in } U,$
$label(v) = label(u)\}$
of V and U, such that
there does not exist $<vi,uj>$
in P such that
there exist vk and ul
such that
vk is a son of vi,
$<vk,ul>$ *is in P, and*
ul is not a son of uj

And here comes the magic - the Prolog code is a direct encoding of the above equivalence and is certainly more readable than it (although perhaps not in the format required in these proceedings):

Algorithm 3:
Story isomorphism

```
isomorphic(story(Sets1,SR1),
           story(Sets2,SR2)) :-
 matchpairs(
           Sets1,Sets2,Pairlist),
 not((member((X,Y),Pairlist),
      member(partition(A,X),SR1),
      member(partition(B,Y),SR2),
   member(Set1,A),
   member((Set1,Set2),Pairlist),
   not(member(Set2,B)))).
```

[5]For the sake of brevity I will call such a forest a *dl-forest*.

```
matchpairs([],[],[]).
matchpairs(
   [(Set1,N)|Rest1],
   Sets2,
   [(Set1,Set2)|Restpairs]):-
 member((Set2,M),Sets2),
 equalsymb(N,M),
 remove((Set2,M),Sets2,Rest2),
 matchpairs(Rest1,Rest2,
                  Restpairs).
```

Note: *member*, *equalsymb* (which tests for symbolic equality) and *remove* are predicates included in the utility library, and used extensively throughout FAME. Note that neither the logical formula nor the algorithm are restricted to dl-forests that contain only "flattened" trees.

2.3 Performance

FAME I was debugged on the original problem

```
(R+1)*C(N,R+1) = (N-R)*C(N,R)
```

and some easier problems. The follwoing is an I/O account of the program's behavior on the above problem.

```
| ?- prove_equal(
|      (r+1)*c(n,r+1),
|      (n-r)*c(n,r)).

The expressions are equal;
stories created for both
have isomorphic canonical forms.


FIRST STORY:

Sets:
Name: set9, Cardinality: n
Name: set7, Cardinality: 1
Name: set8, Cardinality: r+1
```

```
Set relations:
subset(set8,set9)
subset(set7,set8)

AND ITS CANONICAL FORM:

Sets:
Name: set11, Cardinality: r+1-1
Name: set10, Cardinality: n-(r+1)
Name: set9, Cardinality: n
Name: set7, Cardinality: 1

Set relations:
partition(
        [set7,set11,set10],set9)

SECOND STORY:

Sets:
Name: set14, Cardinality: n
Name: set15, Cardinality: r
Name: set12, Cardinality: 1
Name: set13, Cardinality: n-r

Set relations:
partition([set15,set13],set14)
subset(set12,set13)

AND ITS CANONICAL FORM:

Sets:
Name: set16, Cardinality: n-r-1
Name: set14, Cardinality: n
Name: set15, Cardinality: r
Name: set12, Cardinality: 1

Set relations:
partition(
 [set12,set16,set15],set14)
```

Without any further debugging FAME I proved the following equalities:

```
(R+1)*C(N,R+1) = N*(N-1,R)
(N-R)*C(N,R)   = N*(N-1,R)
C(R+R,R)*C(N,R+R) =
       = C(N,N-R)*C(N-R,R)
C(R+R,R)*C(N,R+R) =
         = C(N,R)*C(N-R,R)
N*C(N-1,R-1) = C(N,R)*R
C(N,K)*C(N-K,R-K) =
           = C(N,R)*C(R,K)
```

It also did not mistakenly prove any of a number of wrong equalities given to it as input. In fact, it straightforward to prove that fame is sound for all domains of expressions which are of interest. On the other hand FAME I is not complete for those domains. First, FAME I was not developed to the point where it could reason by case analysis, i.e. create stories for expressions that include operators other than *. Consequently, the program will fail to prove an equality like C(4,1) = C(2,1)*C(2,1). So if X or Y in C(X,Y) are allowed to be strictly numerical, FAME I is not complete. However, if such X and Y are disallowed, the question of FAME I's completeness remains open for now, though my guess is that it is.

2.4 Conclusions and further research

On the face of it, it looks as if I could be happy with the program. Having solved one class of problems I could now extend the program to handle equalities of expressions that are not only a simple product of terms. For example, it wouldn't be too hard to allow some expressions that contain simple sums such as C(2n,2) = 2*C(n,2)+n**2 or some expressions involving summation such as e.g. SIGMA[i from 0 to n: C(n,i)**2] = C(2n,n) (although the latter problem already adds significant difficulty in extending the program).

The real problem begins when you try to extend the program to deal with the partition problems in particular, or counting problems in

general. It was in doing so that I realized that FAME I had a wrong level of knowledge. The correct procedure for story construction should have been to analyze an existing knowledge of counting, and based on that knowledge devise a counting problem appropriate to the given expression. This knowledge is only implicit in FAME I, and therefore cannot be applied differently than it is. In particular there is no natural way to extend FAME I to solve problems 2, 3 and 4.

This was the background for laying FAME I to rest and starting work on FAME II. FAME II was and is being built so as to embody directly knowledge about counting. As mentioned in the Overview, FAME II will be described elsewhere. To give the reader some idea of where things stand at the moment, Figure 3 contains a sample of counting problems currently solved by FAME II:

```
story([(people,100),(team,11),(captain,1),(goalee,1)],
    [subset(team,people),subset(goalee,team),subset(captain,team)]).

or in English: In how many ways can you choose a team of 11 out of
100 candidates, and nominate a captain and goalee (who could be the
person)?.

story([(men,n),(women,r),(god,1)],
    exist(gender,[men,women],[subset(god,gender)])).

or in English: If god is known to be either a man or a woman, and there
are n men and r women, how many different people could it be?

story([(sweets,n),sset(littlekids,r),(poisoned,k)],
    [subset(poisoned,sweets),partition(littlekids,sweets)]).

comment: FAME II transforms the partition problem to alternation
of quantifiers over the stories.

story([(set1,n),(set2,r),(set3,1),ptype(set4,sset1,k),sset(sset1,l),
     sset(sset2,m)],
    exist(formset,sset1,[subset(set3,set1),subset(formset,set2),
                         partition(sset2,set1)])).

story([(set1,5),set2],[subset(set2,set1)]).

comment: notice that the cardinality of set2 is not specified.
```

Figure 3

3 Why Prolog

Conciseness of programming languages has been praised in the literature. The high "idea/symbol ratio" enhances the conceptualizing power of the language. Contrary to APL, however, Prolog achieves the conciseness not through a rich library of system primitives but through a conceptually powerful interpreter. Whatever the correct reason may be, it remains empirically true that Prolog code is short. Although I do not have a LISP program similar, say, to FAME I for a comparison, it is a safe guess that if such a program were written it would be longer than FAME I's length of less than 200 lines.

One reason for the conciseness of the program is the logic programming aspect of Prolog. The dl-forest isomorphism algorithm given in the previous section is a demonstration of that. Another example is the following code for finding the *topsets* of a story (unfortunately, the format of these proceedings makes the code less presentable):

```
topsets(Sets,SR,Topsets) :-
 findall(
  X,
  (member((X,_),Sets),
   not(
   (member(partition(Y,_),SR),
    member(X,Y))),
    not(member(subset(X,_),SR))),
  Topsets).
```

Note: My *findall* predicate behaves like *bagof*, only it does not keep the bindings of the variables between different answers. For more details see [10]).

The logic programming aspect of Prolog is valuable beyond contributing to the conciseness of the code. It was mentioned at the beginning that problem solving shares many features with planning, including self monitoring and making decisions about resource allocation. It was also mentioned that planning depends on a structured domain. In both FAME I and FAME II the planning aspect is at times finessed until the representation is well worked out. In those parts Prolog serves as a default control structure. To use Kowalski's terminology [6], Prolog frees me to work out the logic of the task before fully dealing with the control. Psychologists and AI people talk about *declarative* versus *procedural* knowledge. It is much easier to convert the former to the latter than vice-versa, and to a large extent Prolog facilitates this conversion. For some of the code the declarative interpretation is an artificial one, for indeed it was written with the procedural one in mind. Like in examples given by Gelernter [3], sometimes it is easier to specify the procedure than to specify the outcome. However, these procedures call upon islands of declarative knowledge, which indeed is the case with the way human thinking seems to operate.

For a language to fully qualify as a logic programming language, it should be sound and complete for all first order theories. Prolog clearly does not qualify, since it diverges on even some straightforward theories. In the case of FAME I, for example, the two arguments of the predicate *create_story/2* are an expression and a story, such that the expression is a solution to the story. FAME I uses the former as input and the latter as output, but in a true LP environment the roles would be reversible and *create_story* would actually be a procedure for solving counting problems. The simple minded depth first search algorithm of the Prolog interpreter of course does not provide this luxury, but the user can compensate for this deficiency. Sometimes one can combine the procedural interpretation of the program with the declarative. Most predicates are not "safely reversible", as in the example of the predicate *remove* defined by

```
remove(X,[X|Y],Y).
remove(X,[Y|Z],[Y|W]) :-
 remove(X,Z,W).
```

The goal remove(1,X,Y) diverges without generating all correct answers. Yet sometimes if a particular predicate at a particular point in the code will always have a particular argument instantiated, this otherwise diverging predicate becomes "complete", to misuse the terminology. I term this anchoring the occurrence of a predicate. This is the case with *remove*: if the last argument is guaranteed to be instantiated, the backtracking will uncover all solutions. Thus one conceptually builds mini-theories for code fragments, for which he can safely pretend that Prolog is complete. Such is the case in several modules of FAME I, including the predicates whose code is given in this paper: *pad*, *isomorphic* and *topsets*. All occurrences of *member* and *remove* in them are anchored, and the predicate *equalsymb* is itself reversible.

To summarize, some problem-solving is actually done by the interpreter, in the true spirit of logic programming. In others parts, the interpreter only serves as a default control structure, until a better one is worked out. Finally, at times the interpreter performs a task no different from that of a LISP interpreter, when the control is explicitly and rigidly specified by the programmer. All three uses of Prolog are acceptable, and it is their combination that makes it a powerful and flexible tool.

Acknowledgements This work was closely supervised by my advisor Drew McDermott. Thanks go to him and Tom Dean for comments on earlier versions of the paper.

References

1. John R. Anderson, Games G. Greeno, Paul J. Kline, David M. Neves. *Acquision of problem-solving skill.* In John R. Anderson (ed.), *Cognitive skills and their acquisition*, Lawrence Erlbaum Associates,1981,191-230.

2. Robert S. Boyer, J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

3. David Gelernter. *A note on systems programming in concurrent Prolog.* Proc. of the 1984 Intl. Symp. on Logic Programming (to appear).

4. R. Wm. Gosper. *Decision procedure for indefinite hypergeometric summation.* Math. Review, 58, #5497 (1979).

5. Elaine Kant and Allen Newell. *An automatic algorithm designer: An initial implementation.* Proc. AAAI(83),177-182.

6. Robert Kowalski. *Algorithm = Logic + Control.* Communications of the ACM, July 1979, Vol. 22, #7.

7. *MACSYMA reference manual,* Version 10, 1983. The Mathlab

Group, Laboratory of Computer Science, MIT.

8. David A. McAllester, *The theory and practice of mathematical ontology: A proposal.* PhD thesis, AI laboratory, MIT. (in preparation).
9. Drew V. McDermott. *Planning and acting.* Cognitive Science 2(2), 1978.
10. Fernando Pereira. *Setof again....* A note to the ARPANET Prolog Digest, Vol.1: Issue 45.
11. Earl D. Sacerdoti *A structure for plans and behavior* TR-191, SRI AI center, 1980.
12. Ehud Y. Shapiro. *Algorithmic program debugging.* ACM distinguished dissertation 1982, MIT Press.
13. Gerald J. Sussman, *A computer model of skill acquisition* AI TR-297, AI laboratory, MIT, 1973.
14. M. H. Van Emden, R. A. Kowalski. *The semantics of predicate logic as a programming language.* JACM, 23(4), 1976, pp. 733-742.
15. Stephen Wolfram. *SMP reference manual*
16. Doron Zeilberger. *All binomial equalities are simple.* Proc. National Academy of Science (1982).

A MYCIN-LIKE EXPERT SYSTEM IN PROLOG

Alan Littleford
Prime Computer Inc.
500, Old Connecticut Path
Framingham, MA 01701

ABSTRACT

A large expert system, based on an extended Mycin-like rule set and interpreter, has been constructed using Prolog. The expert system diagnoses computer system crashes in a customer/field engineering environment.

We describe the way in which the expert system is constructed and compare its implementation to other schemes for implementing Mycin-like systems in Prolog. While Prolog itself may be regarded as a production rule system, we still find it necessary to add an extra level of interpretation to satisfy some of our needs.

The system described is in engineering test at Prime Computer, Inc.

1 Backgound

Doc is an expert system which determines the cause of a computer system crash by static analysis of the memory and registers in the system at the time of the crash. Currently this data is held on magnetic tape. The usual procedure has been to let a team of hardware/software experts examine the tape using a simple pretty printer. The analysts would then attempt to determine the cause of the crash and recommend fixes.

Problems handled in this way are usually transient hardware or software problems which are not amenable to conventional diagnostic techniques.

Doc's goal is to substantially reduce the turnaround time for many of these tapes by performing the analysis without human assistance, and

to make fix recommendations wherever possible.

2 Nature of the Problem Domain

An intermittent system crash can be due to many causes - subtle design errors in system code, errant peripherals, environmental conditions, chip failures and configuration errors to name but a few. Often the hardest question to answer about a crash is whether the problem is caused by hardware, software or both.

There are two basic classes of diagnostic expert systems (see Stefik et al. 1982 for a taxonomy of expert systems). The first class includes those systems which are given a description of the structure and correct function of the system, and, reasoning from first principles, deduce the cause of failure (e.g. Genesereth 1982, Davis et al. 1982). The second class of expert systems rely on large numbers of rules causally linking syndromes with probable faults (Shortliffe 1976, Hartley 1984 e.g.).

Those systems which reason from first principles require complete functional descriptions of the system under test. Further, they often need results from dynamically determined tests which are applied as the analysis continues. To give a complete functional description of the hardware and software of a large scale computer system currently is impractical. Further, since we only have the memory image of the crashed system, we are unable to perform experiments to aid diagnosis.

Using syndrome/fault rules seemed promising. The Customer Service group at Prime Computer have, collectively, a large body of rules that are used every day. Another body of knowledge is contained in the manufacturing organisation which is responsible for bringing the system up for the first time and yet another source of expertise is to be found with the design engineers who debug prototype systems.

Problems with this ap-

proach appear when one considers linkage of specific hardware or software knowledge to specific implmentations of the Prime architecture or specific releases of the operating system. This linkage is inevitable and its consideration led to some design choices in the implementation of the expert system. We had to be able to highlight facts or knowledge specific to certain hardware/software configurations, and we had to produce a "language" usable by a diverse collection of knowledge engineers which at the same time would allow those skilled individuals any expressive power they required.

We chose the syndrome/fault approach as a basis for Doc. Specifically we studied the Mycin program (Shortliffe 1976) which diagnoses a class of infectious diseases given the results of clinical tests and other data. We chose Prolog as an implementation language for Doc since many of the features which are built into Mycin are already in the Prolog language (pattern matching and search for example). For reasons discussed below, we actually added a rule interpreter on top of Prolog.

3 Backward Chaining

Mycin is an example of a backwarding chaining system. At any given time the system assumes some hypothesis (expressed as an assertion about the contents of a database) and uses a rule interpreter to select IF-THEN rules which would confirm or deny the hypothesis. The IF parts of the rules are usually conjunctions of other assertions, which, recursively might call upon the rule interpreter to prove or disprove them.

If all the conjunctions of the IF part have been satisfied the assertion(s) contained in the THEN part are added to the database. Mycin adds a numerical belief factor (a measure of the truth of the assertion) to the derived facts, which, together with a formal mechanism for combining belief factors, permits the system to 'weigh the evidence'. These belief factors are hard-coded

into the rules as part of the expertise supplied by the knowledge engineer in his representation of a human expert's problem solving skills in the domain in question.

Belief factors may be viewed as a crutch to be used in the absence of total knowledge of a situation. We have found them to be only partially useful in our specific application and future versions of Doc may discard them.

4 Backward Chaining in Prolog

Excluding for the moment the issue of belief factors, it is clear that Prolog is a good choice for representing backward chaining systems. Indeed Prolog itself is an example of a production rule system using Horn clauses as rules. The rule interpreter is nothing other than the depth-first, left-to-right exploration of the search tree adopted by standard Prolog. This observation has been made by others (Clarke and McCabe 1982).

Nonetheless we discovered that this simple approach has some problems when scaled up to what has since become a very large system :

i. Cost of Computation Our system, like many others, required access to an external database (in our case the crash dump tape). This access is very costly in terms of time, and should be minimised. Thus any conclusions reached from the external facts, or perhaps the facts themselves, need to be cached where possible.

Access to the crash dump tape is characterised by comparatively few (tens to a few hundred) non-localised reads. Given that the tape contains several megabytes of data, we did not think it effective to have the whole tape resident in the Prolog database.

A fact may be required several times during diagnosis. If it is not deduceable the first time it is required then we should make a note so the rule interpreter does not try to deduce it the next time it is required.

ii. Maintainability. Our system requires updating as new

releases of the operating system or hardware enter the field or when new error syndromes are discovered. The system maintainers are unlikely to be Prolog programmers, and since there are likely to be several simultaneous maintainers we felt the need to insulate the underlying Prolog dependencies on clause ordering and backtracking from them.

iii. Multiple Conclusions. Many of the rules we developed contain multiple conclusions. The strategies described elsewhere (Clarke and McCabe 1982) do not lend themselves well do this end, the obvious approach being to have multiple clauses with the same body but different heads :

```
f1 :- i1, i2, i3, .., in.
f2 :- i1, i2, i3, .., in.

% 'if (i1 and i2 and ... in)
%     then  (f1 and f2).'
```

This was rejected since it violates the principle of accurately encoding the knowledge - the knowledge to be encoded includes the fact that f1 and f2 are implied by the same set of circumstances; splitting it up into two rules loses this information. Further, it can be the cause of redundant expensive computations. Another option, encoding f1 and f2 in one head, impedes use of Prolog interpreter pattern matching to select clauses.

iv. Tracing. Our system does not require interactive tracing by a user using 'how' and 'why' questions (Davis, Buchanan and Shortliffe 1977). However we did require the execution of Doc be easily reconstructed at any time after a run (so we could analyse why Doc failed to make a correct diagnosis at our leisure). Thus we require data to be logged concerning why a rule was invoked, the fact it succeeded or why it failed. Using simple Prolog clauses would require each rule take the responsibility of logging its progress and usage, thereby complicating the rule encoding process.

Considering all of the above, we decided to formalise the inference rules by imposing constraints on their form, and interpreting them in an interpreter which would support all logging functions and provide mechanisms for extended control of execution if required.

5 Implementation

5.1 Fact Language

Rules are used to assert facts in a database. We chose to represent facts deducable by rules as Prolog ternary terms of the form

```
f_( Object,Value,Weight).
```

Object and Value may be arbitrary Prolog terms, and Weight is an integer between -10 and 10. Objects are used to represent named state or data structures in the target machine, in which case Value is a representation for the data value of the Objects. Additionally, Object may represent a state in the diagnostic process itself, or even an assertion about the current state of the rule interpreter.

Weight is used to convey the belief factor. We follow the Mycin approach except our weights are integers between -10 (corresponding to certain refutation that Object has Value) to +10 (corresponding to certain confirmation that Object has Value). For example:

```
f_(halt_addr,known,10)
```

"It is definite that the halt address of the machine is known."

```
f_(pb_value(N),[4,305],9)
```

"It is almost certain that the pb (instruction address) of process number N is segment 4 word 305."

```
f_(tr_(halt_addr),true,10)
```

"The rule interpreter has already tried to find a value for the halt address."

In practice, it turns out that almost any knowledge which needs to be represented admits a fairly simple representation in this form. What is more, the valued object representation blends well with a rule-encoder's view of the diagnostic process.

5.2 Rules

We retain the IF-THEN form of Mycin rules. We do not allow backtracking between the various IF parts of the rules, but we allow bactracking **within** an IF part.

Rules are binary terms with principal (infix) functor 'rule'. The first argument is

an atom labelling the rule, and the second is a term of the form

(goal1,goal2,....,goaln)

The goals are passed by the rule interpreter as goals to the Prolog interpreter, so if any of the goals are themselves conjunctions of predicates, backtracking will take place within them. The rule interpreter either sucessfully executes all of the goals or one fails, in which case the rule is aborted. The goal - then(Obj, Value, Weight) - is used to construct facts in the system's database. It always succeeds.

Example:

```
is_pb_consistent rule  (
    defis(pb_value(N), Pb),
    (lies_in(Pb, Area),
          executable(Area)
    ),
    then(state(N, pb),
         consistent, 8)
).
```

N is a process number. The rule attempts to find whether the program counter (Pb) for the process is consistent by seeing whether it lies in an area of memory containing executeable code. Note that backtracking is used to advantage in the second goal (there may be many overlapping 'areas' that the code belongs to, one of which is marked 'executeable').

5.3 Interpreter

The rule interpreter is invoked by use of predicates defis(O,V), mightbe(O,V), defisnot(O,V) etc. These call upon the traced_ predicate with bounds checking on the Weight.

Procedurally, traced_ looks first for an appropriate fact in the database. If none is found but appropriate rules have been tried then fail. Otherwise find all rules whose then parts contain an Object which can be unified with the Object in question. For each rule, unify the corresponding Objects with the goal and then execute the rule as described above. Once all appropriate rules have been executed combine the obtained weights on a per value basis (Objects may be multi-valued) and assert the resulting f_ facts into the database.

```
traced_(Obj,Val,Wt) :-
  f_(Obj, Val, Wt),
  !.
```

```
traced_(Obj, Val, Wt) :-
  f_(tr_(Obj, _, _),
  !, fail.
traced_(Obj,Val, Wt) :-
  trigger(Obj,Val_wt_list),
  addup(Obj,Val_wt_list),
  asserta((f_(tr_(Obj),
                  true,10)),
  f_(Obj, Val, Wt).

trigger(Obj, Wtl) :-
  setof(R,deduces(R,Obj),
                  Rules),
  evaluate( Rules, Wtl).
```

The use of Prolog unification during the search for appropriate rules (Unifying the Object in question with the Object in the then part of the rule) allows use of parameters in the rules. This is very useful in our case where a lot of the rules are of the form "find which process (number) was active. for the active process determine ....". An example is given in the rule displayed above.

The facts

```
f_( tr_( Obj), ....)
```

are examples of facts asserted by the execution of the rule interpreter. These and other facts permit a post mortem of the diagnostic run to be performed. Since Prolog unifies Object with a particular instantiation (e.g. a particular process number in the rule above), the system ensures that the rule will always be fired for different Objects but will return the value in the database (or the fact that no fact can be deduced) once an Object has been used for the first time

(Note: since we have a total history of the execution of rules during a diagnostic run, as well as the rules themselves, it is interesting to postulate the existence of an expert system which would regard this database as **its dump tape** and analyse the 'failure' - lack of adequate diagnosis).

The execution of deduces(R, Object) is improved by "precompiling" the Doc rules as they are loaded into the system. For every rule of the form

```
example rule (
   if1,
   if2,
   ...
   ifn,
   then(Obj1,Val1,W1),
   then(Obj2,Val2,W2),
   ...
   then(Objm,Valm,Wm)
).
```

facts of the form

```
f_(trace(Obji),
```

```
      example,10).
f_(tr_(trace(Obji)),
      true,10).
```

are asserted into the database.

This is analagous to the property-list implementation of the updated-by function in the original Mycin. However in Doc this meta-knowledge is expressed explicitly in the base rule language and is available to the knowledge engineer for exploitation.

5.4 Scripts

It turns out that much of the analyst's knowledge is clumped into procedural "scripts". In these procedures the analyst has a set course of action he takes regardless of the consequences (success or failure) of each individual step. A minor variant of the rule interpreter supports a Prolog script, where each goal is executed regardless of the outcome of previous goals in the script. Much of the top-level procedural control in our system is due to the use of scripts :

```
top_level script (

(defis(stop_type,halt),
 defis(hlt_addr,known),
 defis(known_halt,
                Diag)),

(defis(stop_type, halt),
 defis(hlt_addr,unknown),
 defis(unknown_halt,
                Diag)),

(defisnot(stop_type,
                halt),
 defis(hang_diag, Diag)),

 recite(summary_script)
 %% scripts are recited.
).
```

6 Interface to the Crash Tape

Rules, scripts and the interpreter implement the diagnostic logic of the system, but they need to be interfaced to the pretty printer so they can actually get {{data}} about the specific crash.

The pretty printer used by the analysts performs many functions including symbol table lookup, virtual to physical address mapping etc. Because of implementation time constraints we did not want to modify the pretty printer, which uses terminal i/o and a command language to communicate with its user. Thus we decided to have Doc communicate directly with the pretty

printer program. Evaluable predicates were added to the Prolog interpreter to support inter-process communication at the keyboard I/O level. Thus the expert system could "see" and "type" data at the pretty printer keyboard. Strings returned by the pretty printer are parsed using a Definite Clause Grammar (Pereira and Warren 1982), and strings are generated for transmission by the same method. The result is a compact efficient communications medium between the expert system and the crash tape.

7 Some Statistics

A feasibility demonstration prototype was developed in about two man-months. This system did not communicate directly with the crash tape and only addressed a few symptoms of a crash. Succesful demonstration of this prototype led to funding for a production system which has been in development for about twelve months (two people). The system has shown itself capable of diagnosing crashes in five to fifteen minutes (interpreted Prolog, time-sharing one Mip machine). This compares very favorably against human experts.

An analysis of the execution profile shows that much of the execution time is being spent accessing the dump tape and parsing the data returned from the pretty printer. Each Doc rule call takes about 10-20 logical Prolog inferences.

We currently have approximately three hundred rules, many of which concern detailed architectural knowledge of the machine. Of these rules, several are invoked many times during a run with different values in Object parameters. Thus there may well be more than three hundred rule invocations during one diagnostic session. The actual number of Doc rules invoked varies widely from dump to dump.

The main segments of Prolog code are the rule and script interpreter (about 30 Prolog clauses), a support package for address computations and other utilities (100 clauses), and the various

Definite Clause Grammars.

Doc is due to be shipped to the Customer Service group in Prime during May 1984. There it will be used for internal evaluation.

8 Summary

Prolog lends itself very well to the development of expert systems of the Mycin type. The Prolog language bears much of the complexity burden usually involved in the implementation of such systems, freeing the developer to concentrate on the problem domain. Unmodifed Prolog **is** a production rule system, but it was found to be insufficient for our needs. Our approach has retained the procedural flavour of Mycin whilst still enabling the full power of Prolog to be exploited as needed.

We have found this to provide a very good balance functionally and from the point of view of maintaining a large rule-set.

9 References

K.L. Clarke, F.G. McCabe. PROLOG: a language for implementing expert systems, Machine Intelligence 10, J.E. Hayes, D. Michie, Y.H Pao (eds), Ellis Harwood, 1982.

Randall Davis, Bruce Buchanan and Edward Shortliffe. Production rules as a representation for a knowledge-based consultation program, Artifical Intelligence 8, 15-45, 1977.

Randall Davis et al. Diagnosis based on structure and function, Proceedings of Nat. Conf. on AI, 137-142, 1982.

Michael Genesereth. Diagnosis using hierarchical models, Proceedings of Nat. Conf. on AI, 278-283, 1982.

Roger T. Hartley. Crib: Computer Fault Finding through Knowledge Engineering, IEEE Computer, 76-83, March 1984.

Fernando Pereira and David Warren. Definite clause grammars for language analysis - A survey of the formalism and a comparison with augmented transition networks, Artifical Intelligence, 13,

231-278, 1980.

E.H. Shortliffe. Computer based medical consultation: Mycin. New York, Elsevier, 1976.

Mark Stefik et al. The organisation of expert systems, a tutorial, Artificial Intelligence, 18, 135-173, 1982.

PARLOG FOR DISCRETE EVENT SIMULATION

Krysia Broda and Steve Gregory

Department of Computing, Imperial College
London SW7 2BZ, England

ABSTRACT

In the process interaction method of simulation, entities in the real world are modelled by processes which interact when events occur. In particular, a system can be simulated by a network of parallel processes communicating by messages. In this paper we consider the use of PARLOG to program such simulation models, in which real time must be replaced by a central simulated clock.

1 INTRODUCTION

1.1 The communicating processes approach to simulation

Many kinds of system can be modelled by a collection of processes running in parallel and communicating by sending messages to each other through dedicated channels. Each process models some entity in the system being simulated.

The architecture through which messages pass does not concern us here. However, we find it useful to make some assumptions concerning its operation. The first is that messages from one process to another are processed by the receiver in the same order as they were sent. Secondly, we assume infinite buffering capacity on the message channels. Finally, we shall assume that communication times are so small compared with activity times that an approximation to the system can be obtained by neglecting communication times.

Our first two assumptions are quite reasonable, given the use we will make of such networks of processes. That is, we will be using them to model systems working smoothly. The effects of communication difficulties can be introduced into the model, if desired, by adding new processes to delay or lose messages. The third assumption allows us to ignore the communication times when we come to simulate the passage of time.

The communication channels between processes are uni-directional, i.e. messages travel only in one direction. Communication in the opposite direction can take place only in the form of replies to messages. If several channels are merged into a single channel carrying messages, any replies will effectively be automatically distributed to the senders.

This research was supported (in part) by the SERC under grant number GR/B/97473.

1.1.1 Centralized time

If one is interested in assessing overall system behaviour, then the details of local activities can be ignored; it is the interaction of processes in the system which is important. The timing of the activities can be controlled by a single clock process which receives request messages for alarms to be sent to processes at future times. Processes sending these messages are inactive while awaiting replies, which are the alarm signals.

The periods of local activity of an entity in the real system thus coincide with the inactivity of the process modelling the entity. As far as the global system effect is concerned, such a process is inactive since it is not sending messages.

1.1.2 Simulated time

There are likely to be periods when all processes are inactive. We can achieve a reduction in the run time of the model by skipping such periods. The clock process does not now rely on a real clock to time its issue of alarm reply signals, but continually updates the current time to be that at which the next alarm signal has to be sent. For this to be done, it is essential that the time ordering of events is preserved.

This means that all repercussions of sending an alarm signal must be finished and the system settled down to an inactive state again before the clock can update the time and issue the next alarm signal. The clock process is thus active at exactly those times when all other processes are forced to be inactive.

The result of this alteration is a discrete event process interaction model. It uses the "communicating processes" approach as opposed to, for example, the coroutining approach of SIMULA (Birtwistle et al. 1973). In section 3 we shall consider how to program such models in PARLOG.

1.2 Types of simulation model

Simulation models can be of two main types, continuous and discrete, depending on whether the simulated time is changing continuously or in discrete jumps. One kind of discrete simulation is the fixed time increment, or time advance, approach. The other kind is the discrete event, or event advance, method. See (Emshoff and Sisson 1970) for an introduction to simulation modelling.

In discrete event simulation (Fishman 1978), the simulated time is updated, and hence the system state changed, when an event occurs. One change often causes a chain of events to occur, the time difference between events being negligible. In a discrete event simulation all events in such a chain are considered to occur simultaneously.

There are several approaches to discrete event simulation. The one we take is process interaction, in which the events occur at those times when processes interact. Processes may correspond either to activities or to objects in the real system.

Process interaction models, especially using the communicating processes approach (in which processes interact by messages), can be described graphically. This advantage is exploited when writing a corresponding PARLOG program. Processes are modelled by

PARLOG relations, communication channels by PARLOG streams.

Languages usually employed in writing process interaction models include SIMULA, SIMSCRIPT and GPSS, which are all described in (Fishman 1978). Smalltalk (Goldberg and Robson 1983) has also been used for the same purpose. These are all procedural languages and require expertise from the programmer to design the final program since the interactions between processes must be explicit. In the PARLOG approach, the interactions between processes are implicit and result from the sending of messages. (Although Smalltalk uses message passing, the Smalltalk method of simulation is more similar to that of SIMULA than to the PARLOG method described here.)

1.3 Example: car wash simulation

Consider the problem of modelling a car wash which employs three workers who are all continuously available for work. It takes one worker to wash each car, so the car wash can service up to three cars at a time. Cars arrive at random intervals and enter a line where they wait. As soon as there are workers available, cars are removed from the line on a first-in first-out basis and admitted to the car wash. After a car has been washed (which takes 10 minutes) it leaves the system. Customers of the car wash are not prepared to wait indefinitely for service, so after waiting for some minutes in the line a car will "give up" and leave the system.

Our first step in constructing a communicating processes model for this problem is to draw the graph in Figure 1. Each node is a process and each arc is a communication channel, carrying a stream of messages in one direction as indicated by the arrow. There is no limit to the number of messages that can build up on a channel.

gcargen is a process generating a stream of **CAR** messages on the **arrive** channel from which they are "consumed" by **escapeline.** A **CAR** message being consumed by the **escapeline** process models a car entering the line. When a car gives up and exits from the line, the corresponding **CAR** message is generated on channel **depart2.** **depart2** is merged with **depart1,** the stream of cars leaving the car wash, into **depart** which leads to the **outside** world. The **depart** channel contains the same **CAR** messages as **arrive** though not necessarily in the same order.

The **dcarwash** (meaning "demand carwash") process models the car wash. It represents an activity in which both cars and workers are involved: one worker is required to wash each car. **workers** is a queue representing the pool of (initially three) idle workers. One **WORKER** message is taken from the front of the queue to wash each **CAR** and put back on the rear when finished. The **enter** channel is a stream of **CAR** messages representing cars entering the car wash from the line. Notice that the arrow on this channel points in the opposite direction from the flow of cars. In fact the arrow indicates the flow of demand: requests are sent from the car wash to the line and cars are sent back in reply.

The remaining processes in the graph, **merge, random** and **clock,** are utility processes. **merge** simply interleaves its two input streams in time dependent order. **random** supplies random numbers on demand. **clock** is a process which maintains the system

clock. Any processes which need to delay for a period of simulated time send **HOLD** messages to the clock and wait for an **ALARM** reply.

The graph in Figure 1 itself could comprise part of a program to perform the simulation. To complete the program we need to define each of the processes in the graph. As we shall see in section 3, PARLOG can be used to define the graph and its constituent processes. This is because PARLOG programs have a natural interpretation in terms of networks of communicating processes.

2 OVERVIEW OF PARLOG

PARLOG (Clark and Gregory 1984) is a parallel logic programming language featuring both and- and or-parallelism. For the examples in this paper we need to use only the and-parallel subset of PARLOG, which we shall briefly outline in this section. This language, based on Horn clauses, differs from PROLOG in two crucial respects: "don't care non-determinism" and the use of "modes". These features make possible the concurrent evaluation of conjoined relation calls, i.e. and-parallelism, with stream communication between the calls. Each relation call is evaluated as a process, shared variables act as one-way communication channels along which messages are sent by incremental binding to lists.

The techniques described in this paper could also be applied to Concurrent PROLOG (Shapiro 1983). Concurrent PROLOG uses "read-only" variables instead of modes, so programs do not necessarily have a direct graphical interpretation.

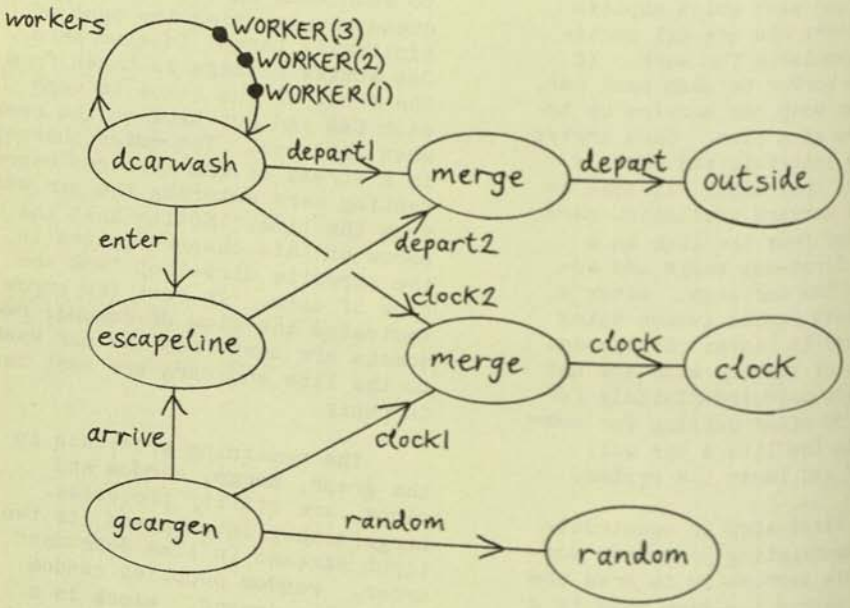


Figure 1: a car wash simulation

2.1 Don't care non-determinism

A PARLOG clause consists of a head atom, a guard conjunction and a body conjunction:

```
H :- G1,...,Gm | B1,...,Bn        (1)
```

The | separates the guard from the body and is omitted if m=0. The , is a parallel "and". | is also read as "and".

In the evaluation of a relation call **r(t1,...,tk)**, all of the clauses for relation **r** will be searched in parallel for a candidate clause. (1) is a candidate clause if the head **H** matches the call **r(t1,...,tk)** and the guard **G1,...,Gm** succeeds, otherwise it is a non-candidate. If all clauses are non-candidates the call fails, otherwise one of the candidates is selected and the call is reduced to the substitution instance of its body **B1,...,Bn**. There is no backtracking on the choice of candidate clause. We "don't care" which candidate clause is selected. In practice, the first one (chronologically) to be found is chosen.

2.2 Modes

Every PARLOG relation definition is preceded by a mode declaration which states whether each argument is input (?) or output (^). For example, the relation **merge(x,y,z)** in the mode to merge lists **x** and **y** to list **z** (lower case identifiers are variables):

```
mode merge(?,?,^).
merge([u|x],y,[u|z]) :-
  merge(x,y,z).
merge(x,[v|y],[v|z]) :-
  merge(x,y,z).
merge([],y,y).
merge(x,[],x).
```

Concurrently evaluating relation calls communicate via shared variables; the modes impose a direction on this communication. In matching a relation call with the head of a clause, there might be an attempt to bind an input variable, i.e. a variable in an input argument of the call. In this case, the attempt to select that clause as a candidate will suspend until some producer further instantiates the variable, eventually becoming either a candidate or a non-candidate. If all clauses for a call are suspended, the call suspends.

3 SIMULATION IN PARLOG

3.1 Example

Let us consider a simplified version of the car wash problem described in section 1.3. Cars arrive in the system at random intervals and enter a first-in first-out queue from which there is no escape. Each car leaves the queue and enters the car wash when a worker is available. After 10 minutes in the car wash the car leaves the system and the worker again becomes available for work. There are initially three idle workers. We wish to simulate the system for 100 minutes.

This time the line of cars waiting to enter the car wash is a simple unbounded FIFO queue. Since a channel has these characteristics, we can represent the line by a channel instead of an explicit process. We start by drawing a graph as in Figure 2.

PARLOG queries can be interpreted as graphs in which processes and channels correspond to relation calls and shared variables respectively. Interpreting a shared variable **v** as a channel, the producer is the output argument where **v** appears. Input

arguments in which **v** appears are consumers. Figure 2 is equivalent to the PARLOG query

```
: cargen(arrive,random,clock1),
  carwash(arrive,
          [WORKER(1),WORKER(2),
           WORKER(3)|workers],
          depart,workers,clock2),
  outside(depart),
  random(random),
  merge(clock1,clock2,clock),
  clock(clock).
```

The types of the variables in this query are shown in Table 1. A **?** annotation marks variables which will be instantiated by the consumer, so that the type includes the direction of communication. Each variable in the above query is a list of terms: the messages sent along the channel. The **?**-annotated variables in Table 1 are the replies. For example, **random** is a list of messages **NEGEXP(a,rn)** where **rn** is a variable to be bound to a number by the consumer of the message. This number will be drawn from a negative exponential distribution with parameter **a**.

workers:	list of **WORKER(id)**
arrive, depart:	list of **CAR(id)**
random:	list of **NEGEXP(a,rn?)**
clock1, clock2, clock:	list of **HOLD(delay,alarm?)**
alarm:	**ALARM(time)**
a, rn, delay, time:	number

Table 1

We have already said that simulation processes send **HOLD** messages to the **clock** process in order to delay for a period of time. These messages are terms of the form **HOLD(delay,alarm)** in which **delay** is the length of the required delay in minutes. **alarm** is a variable which will be bound by the clock, after the required time has elapsed, to the term **ALARM(time)** where **time** is the current time. The sending process

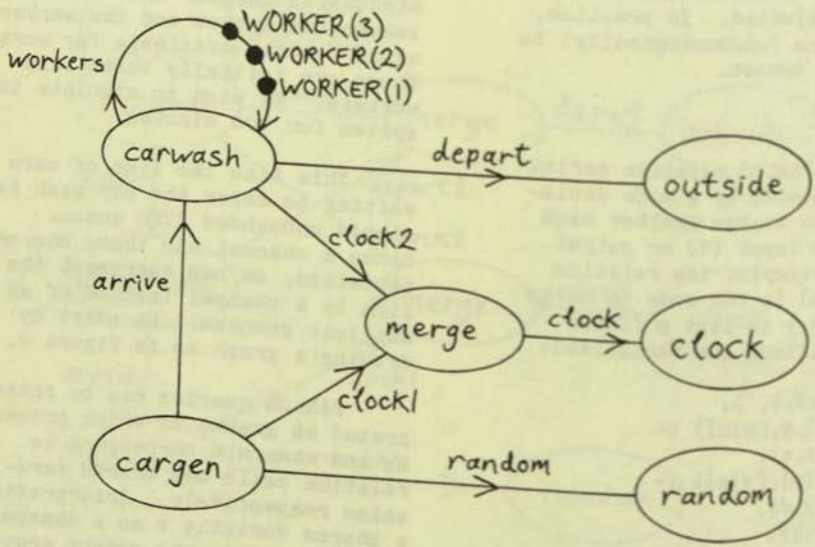


Figure 2: a simple car wash simulation

simply waits for the **alarm** variable to be instantiated.

The **cargen** definition illustrates the use of time delay. Its use as a process is to generate a list of **CAR** messages beginning at time 0 and continuing at random length intervals until 100 minutes has elapsed. Each time it generates a **CAR** message, it also sends a **HOLD** message to the clock and a request for a random number. It then suspends until the **ALARM** reply arrives. The communication pattern is depicted in Figure 3. The dotted lines show the communication from **clock** to **cargen** and from **random** to **clock**.

The definition of **cargen** follows. The first part of the definition acts as initialization.

```
mode cargen(^,^,^).
cargen(arrive,random,
       [HOLD(0,alarm)|clock]) :-
  cargen1(1,alarm,arrive,random,
          clock).
```

```
mode cargen1(?,?,^,^,^).
cargen1(c,ALARM(time),
        [CAR(c)|arrive],
        [NEGEXP(0.09,rn)|random],
        [HOLD(rn,alarm)|clock]) :-
  lesseq(time,100) |
  cargen1(c+1,alarm,arrive,
          random,clock).
cargen1(c,ALARM(time),[],[],[]) :-
  less(100,time) |.
```

We can think of the car wash described earlier as comprising up to three concurrent "washing" activities, each involving a car and a worker. This is modelled by the **carwash** process which matches arriving cars with idle workers and starts a new **wash** process for each **CAR**, **WORKER** pair:

```
mode carwash(?,?,^,^,^).
carwash([CAR(x)|arrive],
        [WORKER(y)|inw],depart,
        outw,clock) :-
  wash(CAR(x),WORKER(y),depart1,
       outw1,clock1),
  carwash(arrive,inw,depart2,
          outw2,clock2),
  merge(depart1,depart2,depart),
  merge(outw1,outw2,outw),
  merge(clock1,clock2,clock).
carwash([],inw,[],[],[]).
```

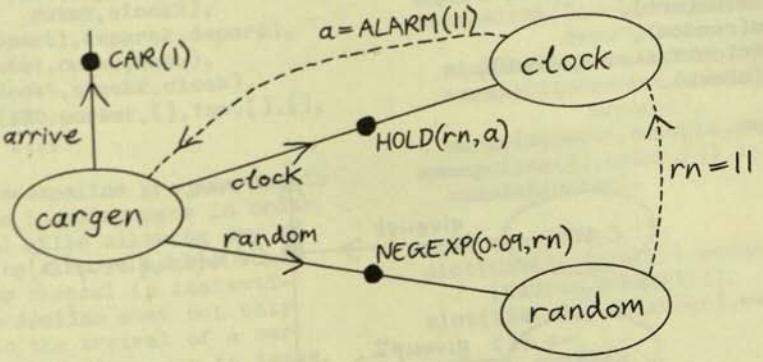


Figure 3: communication between processes

The **wash** process starts by delaying for 10 minutes. When this period has elapsed, the participating **WORKER** is sent back to the pool and the **CAR** is allowed to depart:

```
mode wash(?,?,^,^,^).
wash(CAR(x),WORKER(y),depart,outw,
     [HOLD(10,alarm)]) :-
  wash1(CAR(x),WORKER(y),alarm,
        depart,outw).

mode wash1(?,?,?,^,^).
wash1(CAR(x),WORKER(y),
      ALARM(time),[CAR(x)],
      [WORKER(y)]).
```

3.2 Example

We now return to the problem described in section 1.3, in which waiting cars leave the line after waiting for a certain period of time. The following PARLOG query corresponds to the graph given earlier:

```
: gcargen(arrive,random,clock1),
  dcarwash(enter,
           [WORKER(1),WORKER(2),
           WORKER(3)|workers],
           depart1,workers,clock2),
  escapeline(arrive,enter,
             depart2),
  merge(depart1,depart2,depart),
  outside(depart),
  random(random),
  merge(clock1,clock2,clock),
  clock(clock).
```

The types are as follows, where different from those in Table 1:

arrive, depart1, depart2, depart:	list of car
enter:	list of car?
car:	CAR(<id,giveup>)
giveup:	GIVEUP

In the preceding example, a car was a passive object in the simulation and so could be represented by a message which was a ground term. In some cases, however, we need to simulate objects which have some "intelligence". In the present example a car is no longer passive: it has to decide when to give up and exit from the line. To model this decision-making ability, we now represent each car by a message **CAR(<c,giveup>)** together with a process which has a channel **giveup** into the message, see Figure 4.

The **car** process represents the "intelligence" of a car. It has a **patience** parameter: the maximum length of time it will wait in the line. At the end of this time it will instantiate its **giveup** argument to the term **GIVEUP**.

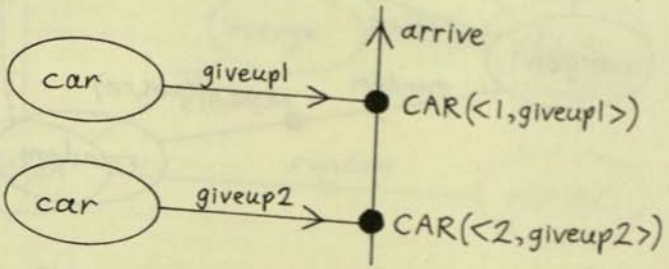


Figure 4: the car message and process

```
mode car(?,^,^).
car(patience,giveup,
   [HOLD(patience,alarm)]) :-
  car1(alarm,giveup).

mode car1(?,^).
car1(ALARM(time),GIVEUP).
```

Logically, the relation **dcarwash** is identical to **carwash**. The difference is that the first argument becomes output instead of input (but it still has the same type: a list of **CAR**s). Behaviourally, the **dcarwash** process generates a list of variables on the **enter** channel. Each time it generates a variable message, it then waits for the variable to be instantiated to a **CAR** term by **escapeline**.

```
mode dcarwash(^,?,^,^,^).
dcarwash([car|enter],
        [WORKER(y)|inw],depart,
        outw,clock) :-
  dcarwash1(car,WORKER(y),enter,
          inw,depart,outw,clock).

mode dcarwash1(?,?,^,?,^,^,^).
dcarwash1(CAR(x),WORKER(y),enter,
        inw,depart,outw,clock) :-
  wash(CAR(x),WORKER(y),depart1,
      outw1,clock1),
  dcarwash1(enter,inw,depart2,
          outw2,clock2),
  merge(depart1,depart2,depart),
  merge(outw1,outw2,outw),
  merge(clock1,clock2,clock).
dcarwash1(END,worker,[],inw,[],[],
        []).
```

The **escapeline** process has to buffer the incoming cars in order of arrival while allowing any of the waiting cars to escape when its **giveup** channel is instantiated. **escapeline** must not only respond to the arrival of a car and the demand for a car to leave, but must also monitor the **giveup** channel of <u>every</u> waiting car.

To handle this and similar situations, we propose the use of "intelligent data structures" (IDSs, see (Gregory 1980)). By an IDS we mean a dynamic network of processes in which each member of the data structure is held by a separate process, which we shall call a "slot". IDSs are particularly useful for data structures whose behaviour depends on changing properties of their contents, as in the present example.

We illustrate **escapeline** by the sequence of graphs in Figure 5. A **slot** process is created for each **CAR** message entering the line and exists until the car leaves the line. There are two ways for a car to leave, corresponding to the two clauses for **slot**. If the **giveup** channel of the car is instantiated to **GIVEUP**, the car is sent out on the **depart** channel. If a demand arrives on the **enter** channel, the car will be supplied in response. Note that if the **GIVEUP** message comes after the car has left the line, it will have no effect and will be ignored.

The PARLOG definition of **escapeline** follows, along with **slot** and **endslot**.

```
mode escapeline(?,?,^).
escapeline([car|arrive],enter,
          depart) :-
  slot(car,enter,enter1,depart1),
  escapeline(arrive,enter1,
            depart2),
  merge(depart1,depart2,depart).
escapeline([],enter,[]) :-
  endslot(enter).

mode slot(?,?,^,^).
slot(CAR(<c,GIVEUP>),enter,enter,
     [CAR(<c,GIVEUP>)]).
slot(CAR(x),[car|enter],enter,
     []) :-
  car = CAR(x).
```

```
mode endslot(?).
endslot([car|enter]) :-
  car = END.
```

The clauses for **slot** and **endslot** assign a reply value to the input variable **car** by a call of the form **car = END**. If this value were to appear in place of **car** in the clause head, the clause would suspend indefinitely waiting for the variable to be instantiated by another process, because it appears in an input argument.

4 THE CLOCK PROCESS

As we have seen, the clock process is responsible for controlling the timing of the simulation processes. It has two tasks: to accept **HOLD** messages from processes, and to issue **ALARM** replies at appropriate times. Let us see how the clock process might be implemented in PARLOG.

Since we shall be simulating time, we shall keep the current simulated time as a local argument of the clock process. Another local argument is the chronologically ordered list of alarm signals to be sent. This list is analogous to the event list in languages such as SIMULA. We shall implement it as a list of pairs **<eventtime,alarm>** in which **alarm** is a variable which is to be bound to the term **ALARM(eventtime)** when the current (simulated) time reaches **eventtime**.

Our first attempt at defining **clock** is as follows:

```
mode clock(?).
clock(clock) :-
  clock1(0,[],clock).
```

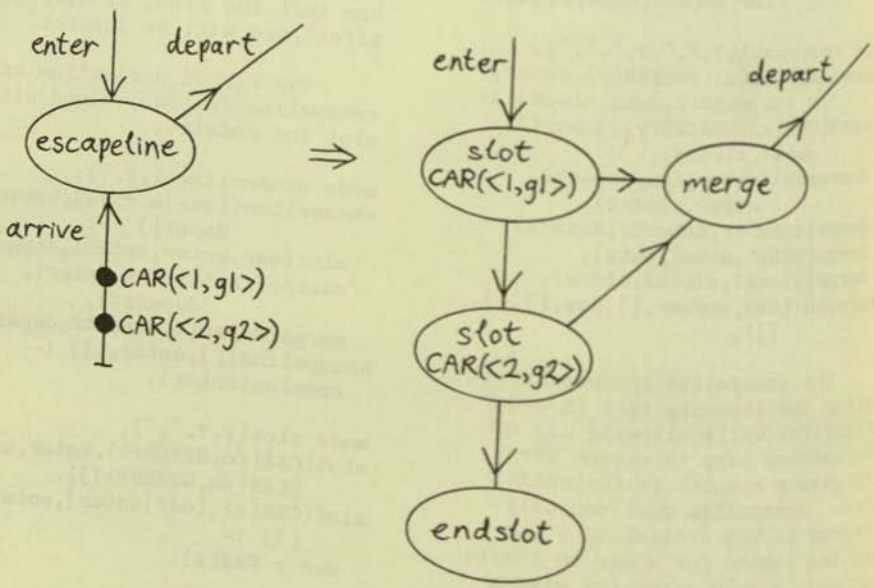


Figure 5: operation of escapeline

```
mode clock1(?,?,?).
clock1(time,[],[]).
clock1(time,events,
    [HOLD(delay,alarm)|clock]) :-
  plus(time,delay,eventtime),
  ordinsert(<eventtime,alarm>,
          events,events1),
  clock1(time,events1,clock).
clock1(time,
      [<eventtime,alarm>|events],
      clock) :-
  alarm = ALARM(eventtime),
  clock1(eventtime,events,clock).
```

The second clause for **clock1** accepts and stores a new **HOLD** message, while the third updates the time to the time of the next event and sends the alarm signal. The problem with this is that there is no restriction on the use of the third clause. As explained in section 1, the clock should only update the time when all simulation processes are suspended, waiting for messages.

We can solve this problem by assuming a primitive PARLOG relation **deadlock**. A call to **deadlock** will suspend until all simulation processes (i.e. processes other than the clock) are suspended, when it will succeed. We change the third clause for **clock1** to

```
clock1(time,
      [<eventtime,alarm>|events],
      clock) :-
  deadlock |
  alarm = ALARM(eventtime),
  clock1(eventtime,events,clock).
```

We shall say more about the **deadlock** primitive in section 5.

5 CONCLUDING REMARKS

5.1 Graphical simulation programs

Previous sections have outlined the PARLOG approach to process interaction simulation modelling. The aim has been to show how a network of communicating processes can be used to model a system by process interaction, and how that graph can easily be realized as a PARLOG program.

We intend to develop a graphical user interface to PARLOG. This would allow the user to develop a program graphically and automatically transform it to a PARLOG program. In fact the graphical program *is* a PARLOG program using a different syntax, so the transformation should be straightforward. This graphical front end would be useful in simulation programming, as we have seen.

There are some special purpose simulation systems to which graphical user interfaces have been added. These are often based on activity scanning where the nodes of the graph represent activities or queues. They however seem to require quite a complicated translation process to turn the graph into a real program.

5.2 Statistics and tracing

For simplicity we have ignored these two important aspects of simulation programs.

Statistics can be collected by the use of one or more processes which keep various data and update it when informed of events by messages. Often, the updating messages need to include the current time. Since only the clock process knows the current simulated time, all update messages must either pass through the clock or to some other process which then asks the clock for the current time (by a message). Either way, the clock process is likely to be heavily loaded by extra message traffic. An alternative is to keep the current time as a global

assertion which can be accessed by the statistics process but updated only by the clock.

Tracing can be done similarly, by a process which displays messages informing of events.

5.3 Implementing the clock

In section 4 we showed how the **clock** process could easily be implemented in PARLOG provided we have a primitive to detect when all simulation processes are suspended. In general, the provision of such a primitive is not easy: it implies having some meta-knowledge about the computation. If a PARLOG program is running on a parallel architecture, it is not clear how any one process can know whether all other processes are suspended.

In a centralized implementation of PARLOG, however, it is a simple matter to detect deadlock since the state of the whole evaluation is accessible (see, e.g. the PROLOG implementation of Concurrent PROLOG given in (Shapiro 1983)). A **deadlock** primitive (actually a variant thereof) is provided in a PARLOG system which we have implemented in PROLOG (Gregory 1983). This system has been used to test the simulation examples in this paper.

5.4 Comparisons

An alternative approach to discrete event simulation in logic is described in (Futo and Szeredi 1982). This is T-PROLOG, an extension of PROLOG to include facilities similar to those found in conventional simulation languages. The attraction of this approach is the use of backtracking to automatically modify the model until the simulation exhibits some desired behaviour.

REFERENCES

Birtwistle G.M., Dahl O.-J., Myhrhaug B. and Nygaard K., SIMULA begin. Petrocelli/Charter, 1973.

Clark K.L. and Gregory S., PARLOG: parallel programming in logic. Research report DOC 84/4, Dept. of Computing, Imperial College, London, 1984.

Emshoff J.R. and Sisson R.L., Design and use of computer simulation models. Macmillan, 1970.

Fishman G.S., Principles of discrete event simulation. John Wiley, 1978.

Futo I. and Szeredi J., A discrete simulation system based on artificial intelligence methods. In Discrete simulation and related fields, ed. A. Javor, North-Holland, 1982.

Goldberg A. and Robson D., Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.

Gregory S., Towards the compilation of annotated logic programs. Research report DOC 80/16, Dept. of Computing, Imperial College, London, 1980.

Gregory S., Getting started with PARLOG. Manual DOC 83/28, Dept. of Computing, Imperial College, London. Also Technical memorandum, ICOT, Tokyo, 1983.

Shapiro E.Y., A subset of Concurrent PROLOG and its interpreter. Technical report TR-003, ICOT, Tokyo, 1983.

Logic Programming by Completion*

Nachum Dershowitz** N. Alan Josephson

Department of Computer Science
University of Illinois
Urbana, IL 61801

ABSTRACT

Term-rewriting systems provide a paradigm of computation with particularly simple syntax and semantics. Rewrite systems may also be used to compute straightforwardly by simplifying terms. We show how the Knuth-Bendix completion procedure may be used to interpret logic programs written as a set of equivalence-preserving rewrite rules. We discuss an implementation of the system and potential advantages of our approach.

1. INTRODUCTION

Term-rewriting systems have been widely used for computation in formula-manipulation and theorem-proving systems. Such a system may be used as a simple nondeterministic language possessing convenient mathematical properties (Hoffman and O'Donnell [1982]). Programs are easy to understand, as they have very simple syntax and semantics, based on equalities, with no explicit control.

In this paper we show how term-rewriting systems may be used to compute in more general settings. The *completion procedure* (Knuth and Bendix [1970]) was introduced as a means of deriving canonical term-rewriting systems to serve as decision procedures for given equational theories. The procedure generates new rewrite rules to resolve ambiguities resulting from existing rules that overlap. We show how that procedure may be used to interpret *logic programs* (Kowalski [1974]) written as a set of equivalence-preserving rewrite rules. Prolog (Clocksin and Mellish [1981]) is one successful attempt to combine the generality of predicate calculus with the efficiency of programming languages and heuristic approaches to problem solving. Unlike Prolog, our method is not restricted to Horn clauses and allows one to incorporate equality between terms in a natural way.[1] We show how rewrite-rule methods may be extended to reason about programs in the general first order predicate calculus (a convenient and natural formalism for knowledge representation), using specifications and domain knowledge, themselves expressed as rewrite rules.

In the next section, we describe rewrite systems and discuss computation by simplification. The main section, Section 3, shows how to use the completion procedure for computing in a rewrite-rule programming language. Section 4 describes some implementation issues. We conclude with a discussion of how the procedure may also be used to verify and synthesize recursive programs in that language.

2. FUNCTIONAL PROGRAMMING

A *term-rewriting (rewrite) system* R over a set of terms T is a finite set of rewrite rules, each of the form $l[\overline{x}] \rightarrow r[\overline{x}]$, where l and r are terms in T containing variables $\overline{x}$. Such a rule may be applied to a term t in T if a subterm s of t matches the left-hand side $l[\overline{\sigma}]$ with some substitution $\overline{\sigma}$ of terms for the variables appearing in l. The rule is applied by replacing the subterm s in t with the corresponding right-hand side $r[\overline{\sigma}]$ of the rule, after the same substitution of terms for variables has been made. The choice of which rule to apply where is made nondeterministically from amongst all possibilities. We write

* This work was supported in part by the National Science Foundation under grant MCS 81-09831.

** While on leave at
Department of Mathematics & Computer Science
Bar-Ilan University
Ramat-Gan 52100
Israel

[1] Hansson, *et al.* [1982] describes a non-Horn logic programming scheme.

$t \Rightarrow t'$ to indicate that a term t' in T is derivable from the term t in T by a single application of some rule in R.

For example, the following system differentiates an expression:[2]

$$
\begin{array}{rcl}
D_x x & \to & 1 \\
D_x a & \to & 0 \\
D_x(u+v) & \to & D_x u + D_x v \\
D_x(u-v) & \to & D_x u - D_x v \\
D_x(-u) & \to & -D_x u \\
D_x(uv) & \to & v D_x u + u D_x v \\
D_x(\frac{u}{v}) & \to & \frac{1}{v} D_x u - \frac{u}{v^2} D_x v \\
D_x(\ln u) & \to & \frac{1}{u} D_x u \\
D_x(u^v) & \to & v u^{v-1} D_x u + u^v (\ln u) D_x v
\end{array}
$$

where u and v are variables of the rewrite system and match any term, x is the symbol with respect to which an expression is differentiated, and a is any atomic symbol other than x.

Thus, to find the second derivative of $\frac{1}{x}$, we use the above rules along with rules axiomatizing subtraction, addition, and exponentiation to reduce the term $D_x(D_x \frac{1}{x})$. Applying the rule for terms of the form $\frac{u}{v}$ yields $D_x(\frac{1}{x} D_x 1 - \frac{1}{x^2} D_x x)$. (The numerals used are just abbreviations for their unary representation as sums of ones, e.g. 2 is short for $1+1$.) Rewriting $D_x 1$ to 0 and successively applying the rules $u * 0 \to 0$ and $0 - u \to -u$ (here unary $-$ and subtraction are distinguished) yields the term $D_x(-\frac{1}{x^2})$. Continuing to reduce, we finally get $\frac{2}{x^3}$, which can be rewritten by no other rule. In this manner, rewrite rules have long been used as 'functional programs' for *ad hoc* computation in symbol manipulation systems (e.g. Hearn [1971]). We note that rewrite systems have the full computational power of Turing machines (Huet and Lankford [1978]).

2.1. Termination

A system R is said to *terminate* for a set of terms T if there is no infinite derivation $t_1 \Rightarrow t_2 \Rightarrow t_3 \Rightarrow \cdots$ of terms t_i in T. The standard method of demonstrating termination is to use monotonic well-founded orderings on terms. A survey of orderings useful for proving termination may be found in Dershowitz [1983].

2.2. Superposition

Let $l[\overline{u}] \to r[\overline{u}]$ and $l'[\overline{v}] \to r'[\overline{v}]$ be two (not necessarily different) rules in R whose variables $\overline{u}$ and $\overline{v}$ have been renamed, if necessary, so that they are distinct. We say that l *overlaps* l', if $l[\overline{u}]$ contains a (nonvariable) subterm s embedded in some context t—to indicate this we write $l[\overline{u}] = t[s][\overline{u}]$—such that there is a (most general) substitution $\overline{\sigma}$ for the variables $\overline{u}$ and $\overline{v}$ for which $s[\overline{\sigma}] = l'[\overline{\sigma}]$. If l overlaps l', then the overlapped term $l[\overline{\sigma}]$ can be rewritten to either $r[\overline{\sigma}]$ or $t[r'][\overline{\sigma}]$. These two possibilities form a *critical pair*. During the completion algorithm such pairs become new rules, oriented with respect to ordering $>$.

2.3. Associativity and Commutativity

Associativity and commutativity of functions cannot be handled by including axioms for those properties as rules. Instead, special unification algorithms are used to take associativity and commutativity into account.

As an example, consider the following canonical rewrite system for Boolean algebra:[3]

$$
\begin{array}{rcl}
\sim u & \to & u \oplus \mathbf{true} \\
u \vee v & \to & u \wedge v \oplus u \oplus v \\
u \supset v & \to & u \wedge v \oplus u \oplus \mathbf{true} \\
u \wedge \mathbf{true} & \to & u \\
u \wedge \mathbf{false} & \to & \mathbf{false} \\
u \wedge u & \to & u \\
u \oplus \mathbf{false} & \to & u \\
u \oplus u & \to & \mathbf{false} \\
(u \oplus v) \wedge w & \to & u \wedge w \oplus v \wedge w
\end{array}
$$

where $\sim$ is 'not', $\wedge$ is 'and', $\vee$ is 'inclusive-or', $\oplus$ is 'exclusive-or', and $\supset$ is 'implies'. Both $\wedge$ and $\oplus$ are implicitly associative and commutative. That means, for example, that the rule $u \wedge u \to u$ applied to $(p \wedge q) \wedge p$ yields $p \wedge q$. Since these functions are associative, there is no significance to the parenthesization, and accordingly terms are 'flattened' by removing embeddings of associative functions symbols, e.g. $(p \wedge q) \wedge p$ is written $p \wedge q \wedge p$.

[2] Knuth [1968], p. 337.

[3] Watts and Cohen [1980], Hsiang [1982].

That this system is sound (i.e. terms are rewritten only to equal terms) follows from the fact that each rule is a propositional equivalence and $\wedge$ and $\oplus$ are in fact associative and commutative. The termination of this system can be shown by various methods described in Dershowitz, et al. [1983].

When, as in this example, some of the functions on the left-hand sides l or l' are associative and commutative, then an associative-commutative unification algorithm (Livesey and Siekmann [1976], Stickel [1981], Fages[1984]) is used to find σ such that $l[\sigma]$ and $l'[\sigma]$ overlap. The definition of 'overlap' must also be extended to include cases in which two rules have overlapping subterms of the same associative-commutative symbol (Lankford and Ballantyne [Aug. 1977], Peterson and Stickel [1981]). To do this, pseudo-rules $f(l,u') \rightarrow f(r,u')$ are considered for each rule whose left-hand side l has an associative-commutative outermost symbol f. All such critical pairs must reduce to the same term up to permutation of arguments of the associative-commutative symbols.

3. LOGIC PROGRAMS

Rewrite systems may be used as 'logic programs' (Kowlaski [1974]), in addition to their straightforward use for computation by rewriting. For example, the following is a Prolog-like rewrite program for appending two lists:

$$\begin{array}{rcl} z\cdot w = app(z\cdot u,v) & \rightarrow & w = app(u,v) \\ v = app(\mathrm{nil},v) & \rightarrow & \mathbf{true} \\ v = app(v,\mathrm{nil}) & \rightarrow & \mathbf{true} \end{array}$$

The completion procedure, given this program, the rule $z = z \rightarrow \mathbf{true}$ for equality, and the goal rule

$$w = app(a\cdot[b\cdot[c\cdot\mathrm{nil}]],d\cdot[e\cdot\mathrm{nil}]) \rightarrow ans(w),$$

generates the computation

$$\begin{array}{l} w = app(b\cdot[c\cdot\mathrm{nil}],d\cdot[e\cdot\mathrm{nil}]) \rightarrow ans(a\cdot w) \\ w = app(c\cdot\mathrm{nil},d\cdot[e\cdot\mathrm{nil}]) \rightarrow ans(a\cdot[b\cdot w]) \\ w = app(\mathrm{nil},d\cdot[e\cdot\mathrm{nil}]) \rightarrow ans(a\cdot[b\cdot[c\cdot w]]) \\ \quad w = d\cdot[e\cdot\mathrm{nil}] \rightarrow ans(a\cdot[b\cdot[c\cdot w]]) \\ ans(a\cdot[b\cdot[c\cdot[d\cdot[e\cdot\mathrm{nil}]]]]) \rightarrow \mathbf{true}. \end{array}$$

The programming paradigm described below yields a Prolog-like programming language, the main differences being that rewrite rules are equivalences rather than implications in Horn-clause form, and that the Knuth-Bendix [1970] completion procedure acts as the interpreter, rather than resolution. Hogger [1981] suggested the use of equivalences to specify Prolog programs. The idea of using a (resolution) theorem-prover for (straight-line) computation was suggested by Green [1969] and Waldinger [1969].

Consider the following program for computing the quotient and remainder of two integers:

$$\begin{array}{c} div(u+v+1,v+1,q+1,r) \rightarrow \\ div(u,v+1,q,r) \\ div(u,u+w+1,0,u) \rightarrow \mathbf{true} \\ div(u+v+1,v+1,1,r) \rightarrow \\ div(u,v+1,0,r) \\ div(v+1,v+1,q+1,r) \rightarrow \\ div(0,v+1,q,r) \\ div(v+1,v+1,1,r) \rightarrow div(0,v+1,0,r) \\ div(u+1,1,q+1,r) \rightarrow div(u,1,q,r) \\ div(u+1,1,1,r) \rightarrow div(u,1,0,r) \\ div(1,1,q+1,r) \rightarrow div(0,1,q,r) \\ div(1,1,1,r) \rightarrow div(0,1,0,r) \\ div(u,u+1,0,u) \rightarrow \mathbf{true} \\ div(0,1,0,0) \rightarrow \mathbf{true} \end{array}$$

where $+$ is associative and commutative. The first rule is the main recursive case; the second is the main base case; the third simplifies sums; the remainder are special cases. To compute the quotient and remainder of two numbers a and b with this system, the rule

$$div(a,b,q,r) \rightarrow ans(q,r)$$

is added, meaning that q are r are the answer if and only if they are the quotient and remainder, respectively, of a and b. The completion procedure then generates a rule

$$ans(c,d) \rightarrow \mathbf{true},$$

containing the answer values c and d for q and r, respectively.

3.1. The Completion Procedure

To compute with a rewrite program, the completion procedure is used. In general, the procedure gets as input a finite set R of rules, a finite set E of equations, and a program to compute a monotonic well-founded ordering $>$. Initially, R may contain any set of sound reductions, all of whose reduced critical pairs are in the input set E. (A "reduced" critical pair is a critical pair, both sides of which have been reduced by R.) The procedure then uses superposition to generate new rules, each of which is a sound reduction. (See Huet [1981]

for details.)

If the completion procedure terminates without failure (*i.e.* it was able to orient each newly formed rule), then it returns as output a canonical system R for E. Furthermore, it may be that a particular choice of well-founded ordering $>$ precludes finding a canonical system. (This separation between axioms and ordering corresponds to the competence/performance dichotomy in programming advocated by Pratt [1977] and others.) The procedure may also go on generating an infinite number of new rules without ever finding a canonical system or aborting. In that case, we say that the procedure *loops*.

Theorem 1 (Huet [1981]). *An equation $M=N$ is valid in an equational theory E, if and only if the completion procedure—given the equations E—eventually will have generated enough rules for M and N to reduce to the identical term. This, provided that the procedure does not abort.*

We assume that this result also applies when some of the function symbols are implicitly associative and commutative (see Lankford [1981]), and that associative-commutative unification is possible in the presence of non-associative-commutative function symbols (Fages [1984]). Accordingly, we allow rewrite systems to contain function symbols that are implicitly associative and commutative, and use the extensions of the Knuth-Bendix procedure to commutative functions (Lankford and Ballantyne [Mar. 1977]) and to associative-commutative functions (Lankford and Ballantyne [Aug. 1977], Peterson and Stickel [1981]).

3.2. Computing

To compute by completion, a *goal rule* is added to a rewrite system. Goal rules are of the form $p[\bar{x},\bar{y}] \rightarrow ans(\bar{y})$, where p is the *calling term* containing input values (*i.e.* irreducible ground terms) $\bar{x}$ and output variables $\bar{y}$, and *ans* is the function symbol that will store the result.[4] With the exception of predicates $=$ and *ans* we adopt the functional approach, representing operations as functors and including the rule $z=z \rightarrow$ **true** to effect assignment. We define a *rewrite program* to be a finite set of rewrite rules for which the completion procedure may be applied *linearly*. That is, the goal rule and rules derived from it are only overlapped with the rules of the program. Derived rules are never overlapped with themselves, nor are program rules overlapped with themselves.

Let $P(\bar{x},\bar{y})$ be a predicate on ground terms $\bar{x}$ and $\bar{y}$. A rewrite program R with calling sequence $p[\bar{x},\bar{y}]$ is said to *compute* the *output predicate* $P(\bar{x},\bar{y})$, if, given a goal $p[\bar{x},\bar{y}] \rightarrow ans(\bar{y})$ for ground terms $\bar{x}$, the completion procedure will generate a rule $ans(\bar{t}) \rightarrow$ **true**, such that $P(\bar{x},\bar{t})$ holds (providing such a $\bar{t}$ exists), without aborting. The ordering supplied to the procedure should make **true** less than *ans* terms and *ans* terms less than any other term.

The following theorem provides a sufficient, but not necessary, condition for a rewrite system to act as a program in the above sense.

Theorem 2. *A rewrite program R computes an output predicate P if R is correct with respect to true ground input terms $p[\bar{x},\bar{t}]$ and the constant* **true**.

What this theorem means is that if the rewrite system evaluates ground terms of the form $p[\bar{x},\bar{t}]$ to **true** whenever $P(\bar{x},\bar{t})$ is true, then adding a goal rule $p[\bar{x},\bar{y}] \rightarrow ans(\bar{y})$ to R, where $\bar{x}$ are the input values and $\bar{y}$ are variables, and completing is guaranteed to generate a rule of the form $ans(\bar{t}) \rightarrow$ true such that $P(\bar{x},\bar{t})$, if such a $\bar{t}$ exists.

The division program computes correctly since it reduces ground calling terms of the form $div(u,v+1,q,r)$ to **true** whenever the numerals q and r are the quotient and remainder, respectively, of the numerals u and (the nonzero) $v+1$. For example, to compute the quotient and remainder of 7 and 3, the rule

$$div(7,3,q,r) \rightarrow ans(q,r)$$

is added. Completion generates

$$div(4,3,q,r) \rightarrow ans(q+1,r)$$

by overlapping the goal rule with the first program rule

$$div(u+v+1,v+1,q+1,r) \rightarrow div(u,v+1,q,r);$$

using the same program rule once more gives

[4] The latter is akin to the 'answer literal' of Green [1969].

$$div(1,3,q,r) \rightarrow ans(q+2,r);$$

overlapping this with the second program rule $div(u,u+w+1,0,u) \rightarrow \mathbf{true}$ and applying the simplification $z+0 \rightarrow z$ yields the answer rule

$$ans(2,1) \rightarrow \mathbf{true}.$$

The same program may be used to compute other arguments of *div*. For example, to compute the product of 3 and 2, one adds the goal

$$div(u,3,2,0) \rightarrow ans(u).$$

Completion generates

$$\begin{aligned} div(u,3,1,0) &\rightarrow ans(u+3)\\ div(u,3,0,0) &\rightarrow ans(u+6)\\ ans(6) &\rightarrow \mathbf{true}. \end{aligned}$$

Although any Prolog statement may be directly translated to a single rewrite rule, the converse is not the case. In general, a Prolog statement of the form

$$A \leftarrow B_1, B_2, \ldots, B_n$$

(meaning that A is implied by B_1 through B_n) corresponds to the rule

$$A \wedge B_1 \wedge \cdots \wedge B_n \rightarrow B_1 \wedge \cdots \wedge B_n;$$

Prolog axioms $A \leftarrow$ correspond to $A \rightarrow \mathbf{true}$ and goals $\leftarrow B$ to $B \rightarrow \mathbf{false}$. It is not difficult to see that the linear completion procedure will not abort for any program that consists of rules with only conjunctions of literals (or **true**/**false**) on both sides.

The following rewrite-program specifies an insertion sort:

$$\begin{aligned} sort(\mathbf{nil}) &\rightarrow \mathbf{nil}\\ sort(x\cdot\mathbf{nil}) &\rightarrow x\cdot\mathbf{nil}\\ z=sort(x\cdot u)\wedge y=sort(u)\wedge z=ins(x,y) &\\ \rightarrow z=ins(x,y)\wedge y=sort(u) &\\ ins(x,\mathbf{nil}) &\rightarrow x\cdot\mathbf{nil}\\ x\cdot[y\cdot z]=ins(x,y\cdot z)\wedge x\leq y &\rightarrow x\leq y\\ y\cdot z=ins(x,y\cdot u)\wedge y\leq x\wedge z=ins(x,u) &\\ \rightarrow y\leq x\wedge z=ins(x,u) & \end{aligned}$$

The function $ins(y,z)$ returns the list resulting from insertion of y in its proper place in sorted list z with respect to the primitive predicate $\leq$. ($\leq$ can be defined by the rules $u\leq u \rightarrow \mathbf{true}$ and $u\leq u+v \rightarrow \mathbf{true}$.) The goal rule

$$z=sort(3\cdot[1\cdot[2\cdot\mathbf{nil}]]) \rightarrow ans(z)$$

overlaps with the third rule to produce

$$z=ins(3,y)\wedge y=sort(1\cdot[2\cdot\mathbf{nil}]) \rightarrow ans(z)\wedge z=ins(3,y)\wedge y=sort(1\cdot[2\cdot\mathbf{nil}]).$$

The left-hand side of this rule does not unify with the left-hand side of any of the program rules. When we consider the extension

$$z=sort(x\cdot u)\wedge y=sort(u)\wedge z=ins(x,y)\wedge w \rightarrow y=sort(u)\wedge z=ins(x,y)\wedge w$$

of the third rule, however, we get the overlap

$$z=ins(3,y)\wedge y=ins(1,w)\wedge w=sort(2\cdot\mathbf{nil}) \rightarrow ans(z)\wedge z=ins(3,y)\wedge y=ins(1,w)\wedge y=sort(1\cdot[2\cdot\mathbf{nil}])\wedge w=sort(2\cdot\mathbf{nil}).$$

Continuing in this fashion, superposing program rules with the goal rule, we get

$$ans(1\cdot[2\cdot[3\cdot\mathbf{nil}]]) \rightarrow \mathbf{true}.$$

That is, $1\cdot[2\cdot[3\cdot\mathbf{nil}]]$ is the sorted version of $3\cdot[1\cdot[2\cdot\mathbf{nil}]]$.

3.3. Combining Programming Modes

The two uses of rewrite systems, for straightforward computation by simplification and for computation by completion, may be combined in a single program. If we consider the differentiation program of the section 2, we see that it can also be used to integrate. Thus, to compute the integral of x^2, we add the rule

$$x^2=D_x y \rightarrow ans(y),$$

along with rules for $-$, exponentiation, $*$, and $+$. (The last two are associative-commutative operators and need the related unification algorithm.) Completion generates

$$\begin{aligned} x^2=(D_x u+D_x v) &\rightarrow ans(u+v)\\ x^2=D_x u &\rightarrow ans(u+a)\\ x^2=(yD_x z+zD_x y) &\rightarrow ans(yz+a)\\ x^2=cD_x z &\rightarrow ans(cz+a)\\ x^2=c((v+1)u^v D_x u+u^{v+1}\ln u D_x(v+1)+1) &\\ \rightarrow ans(cu^{v+1}+a) &\\ x^2=c((k+1)u^k D_x u) &\rightarrow ans(cu^{k+1}+a)\\ x^2=c(k+1)x^k &\rightarrow ans(cx^{k+1}+a) \end{aligned}$$

$$x^2=x^k \rightarrow ans(\frac{1}{k+1}x^{k+1}+a)$$

$$ans(\frac{1}{3}x^3+a) \rightarrow \textbf{true}.$$

By performing simplifications whenever possible, the number of new superpositions is greatly reduced.

4. IMPLEMENTATION

TeRSe (Hsiang and Josephson [1983]) is a rewrite rule theorem prover that uses the Boolean algebra system of section 2.3 as the basis for a complete refutational strategy for first order theory. We have implemented the rewrite program paradigm within *TeRSe* for rules containing only $\wedge$ (this includes all Horn-clauses). We have tailored the control mechanism to linear completion and are using the following improvements to gain efficiency within the system.

4.1. Program Control

Since Horn clauses give rise to rules of the form

$$A \wedge B_1 \wedge \cdots \wedge B_n \rightarrow B_1 \wedge \cdots \wedge B_n,$$

we always want to unify the literal A (otherwise, we get a trivial rule containing the answer predicate on both sides). There are other constraints on which arguments to $\wedge$ get unified which lead to similar speedups.

4.2. Redundant Rewriting

Rewriting can be made more efficient by keeping track of which rules have been applied at which levels. Since the effect of a rewrite only changes a local portion of the term, by maintaining a list of occurences, redundant unifications can be eliminated.[5] Control of this sort corresponds to the tight 'inner-loop' of Prolog.

4.3. Assertions about Equality

Unification is not an optimal way for dealing with arithmetic operators. For example, in trying to bind the term $x+3$ (really $x+1+1+1$) with 4 $(1+1+1+1)$, the result $x=1$ will be found only after invocation of the associative-commutative unification for the operator $+$. To take advantage of the semantic meaning of the symbol $+$ requires having *assertions about equality* as described in Kornfeld [1983]. Thus, knowing that $x+a=b \supset x=b-a$, where $-$ is interpreted in the natural sense, obviates the need for unification at the symbol $+$.

[5] Cf. methods in Nelson and Oppen [1980] for congruence closure.

5. DISCUSSION

We have illustrated how rewrite rules are used for general-purpose computation. Each rule is an equality between terms or equivalence between formulas. The result is a nondeterministic programming language that has all the advantages of logic programs, including clean syntax, well-understood semantics, and the ability to use the same language (and not just Horn-clauses) for both specification and computation. Rewrite programs have the additional advantage of allowing rules expressing equality between terms to be incorporated. We have described how the *TeRSe* environment is being used to acheive an efficient implementation of the programming methodology.

Furthermore, the full completion procedure may be used to "compile" a complete program given a partial definition. More generally, completion—like other theorem-proving methods—can be applied to the task of automatic rewrite-program synthesis from specifications. The completion procedure itself does the 'folding' (that is, the introduction of recursive calls) based upon the axiomatization of the problem domain. Specifications are expressed in the same language as programs, with the Boolean algebra system providing the necessary logical capability. (Compare the methodologies of Burstall and Darlington [1977], Clark and Tarnlund [1977], Kowalski [1979], Manna and Waldinger [1980], Clark [1981], and Hogger [1981].) If the completion procedure is given the right ordering then it will find a program, if a program exists, that does not require auxiliary definitions. When auxiliary functions are needed, their definition may be supplied by the user.

Assume that we wish to synthesize a program for some predicate $P(\overline{x},\overline{z})$, and are given an axiomatization E of the problem domain and a set H of equations specifying the required properties of P. We can start the completion procedure off with E and H and run it until a program R is generated that computes the specification P. The monotonic well-founded ordering supplied to the completion procedure should ensure that terms

containing 'specification' symbols are greater than corresponding terms containing the defined goal symbol, which in turn should be greater than **true**. The particular choice of ordering will, of course, affect the program derived. Given an appropriate ordering, the completion procedure will find a program meeting the specifications, unless it aborts. In a similar manner, the completion procedure may be used to verify program correctness (see Dershowitz [Dec. 1982]).

REFERENCES

1. R.M. Burstall and J. Darlington [Jan. 1977], "A transformation system for developing recursive programs," *J. ACM*, vol. 24, no. 1, pp. 44-67.
2. K.L. Clark [Sept. 1981], "The synthesis and verification of logic programs," Research report DOC 81/36, Dept. of Computing, Imperial College, London, England.
3. K.L. Clark and S.-A. Tarnlund [1977], "A first order theory of data and programs," *1977 IFIP Cong. Proc.*, pp. 939-944.
4. W.F. Clocksin and C.S. Mellish [1981], *Programming in Prolog*, Springer-Verlag, Berlin.
5. N. Dershowitz [Dec. 1982], "Applications of the Knuth-Bendix completion procedure," *Proc. Seminaire d'Informatique Theoretique*, Univ. Paris VII, Paris, to appear.
6. N. Dershowitz [1983], "Termination," in *Systems of Reductions* (A.M. Ballantyne and M. Richter, eds.), to appear.
7. N. Dershowitz, et. al. [August 1983], "Associative-Commutative Rewriting," *1983 IJCAI Proc.*, vol. 2, pp. 940-944.
8. F. Fages [Mar. 1984], "Associative-Commutative Unification," Proc. 9th Colloq. on Trees in Algebra and Programming, Bordeaux, France, to appear.
9. C.C. Green [June 1969], *The application of theorem-proving to question-answering*, Ph.D. thesis, Dept. of Computer Science, Stanford Univ., Stanford, CA.
10. A. Hansson, S. Haridi, S.-A. Tarnlund [1982], "Some Aspects on a Logic Machine Prototype," in *Logic Programming* (K.L. Clark and S.-A. Tarnlund, eds.), Academic Press, London, pp. 267-280.
11. A.C. Hearn [1971], "REDUCE 2—A system and language for algebraic manipulation," *Proc. 2nd ACM Symp. on Symbolic and Algebraic Manipulation*, Los Angeles, CA, pp. 128-133.
12. C.M. Hoffman and M.J. O'Donnell [Jan. 1982], "Programming with equations," *Trans. on Programming Languages and Systems*, vol. 4, no. 1, pp. 83-112.
13. C.J. Hogger [Apr. 1981], "Derivation of logic programs," *J. ACM*, vol. 28, no. 2, pp. 372-392.
14. J. Hsiang [Dec. 1982], *Topics in automated theorem proving and program generation*, Ph.D. thesis, Dept. of Computer Science, Univ. of Illinois, Urbana, IL.
15. J. Hsiang, N.A. Josephson, [1983], "TeRSe: A Term Rewriting Theorem Prover," *Proceedings of Rewrite Rule Workshop*, G.E., Schnectady, N.Y.
16. G. Huet [1981], "A complete proof of correctness of the Knuth-Bendix completion algorithm," *J. Computer and System Sciences*, vol. 23, no. 1, pp. 11-21.
17. G. Huet and D.S. Lankford [1978], "On the uniform halting problem for term rewriting systems," Rapport Laboria 283, IRIA, Le Chesney, France.
18. G. Huet and D.C. Oppen [1980], "Equations and rewrite rules. A survey," in *Formal Language Theory: Perspectives and Open Problems* (R. Book, ed.), Academic Press, New York, pp. 349-405.
19. D. E. Knuth [1968], *The Art of Computer Programming*, vol. 1: "Fundamental Algorithms," Addison-Wesley, Reading, MA.
20. D.E. Knuth and P.B. Bendix [1970], "Simple word problems in universal algebras," in *Computational Problems in Abstract Algebra* (J. Leech, ed.), Pergamon Press, Oxford, pp. 263-297.
21. W. A. Kornfeld [1983], "Equality for Prolog," *1983 IJCAI Proc.*, vol. 1, pp. 514-519.
22. R.A. Kowalski [1974], "Predicate logic as programming language," *Proc. IFIP-74 Cong.*, North-Holland, Amsterdam.

23. R.A. Kowalski [1979], *Logic for Poblem Solving*, North-Holland, New York, pp. 204-206.

24. D.S. Lankford [Aug. 1981], "A simple explanation of inductionless induction," Memo MTP-14, Dept. of Mathematics, Louisiana Tech Univ., Ruston, LA.

25. D.S. Lankford and A.M. Ballantyne [Mar. 1977], "Decision procedures for simple equational theories with commutative axioms: Complete sets of commutative reductions," Memo ATP-35, Dept. of Mathematics and Computer Sciences, Univ. of Texas, Austin, TX.

26. D.S. Lankford and A.M. Ballantyne [Aug. 1977], "Decision procedures for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions," Memo ATP-39, Dept. of Mathematics and Computer Sciences, Univ. of Texas, Austin, TX.

27. M. Livesey and J. Siekmann [1976], "Unification of A+ C-terms (bags) and A+ C+ I-terms (sets)," Intern. Ber. Nr. 5/76, Inst. fur Informatik, Univ. Karlsruhe, Karlsruhe, W. Germany.

28. Z. Manna and R.J. Waldinger [Jan. 1980], "A deductive approach to program synthesis," *ACM Trans. on Programming Languages and Systems*, vol. 2, no. 1, pp. 92-121.

29. G. Nelson and D.C. Oppen, [1980], "Fast Decision Procedures Based on Congruence Closure," *J. ACM 27, 2* (April 1980), pp. 356-364.

30. G.E. Peterson and M.E. Stickel [Apr. 1981], "Complete sets of reductions for some equational theories," *J. ACM*, vol. 28, no. 2, pp. 233-264.

31. V.R. Pratt [Jan. 1977], "The competence/performance dichotomy in programming," *Proc. 4th ACM Symp. on Principles of Programming Languages*, Santa Monica, CA.

32. M.E. Stickel [July 1981], "A unification algorithm for associative-commutative functions," *J. ACM*, vol. 28, no. 3, pp. 423-434.

33. R.J. Waldinger [May 1969], "Constructing programs automatically using theorem proving," Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA.

34. D.E. Watts and J.K. Cohen [Aug. 1980], "Computer implemented set theory," *American Math. Monthly*, vol. 87, no. 7, pp. 557-560.

ASSOCIATIVE CONCURRENT EVALUATION OF LOGIC PROGRAMS

Katsuhiko Nakamura

School of Science and Engineering
Tokyo Denki University
Hatoyama-machi, Saitama-ken, 350-03 Japan

ABSTRACT

A general evaluation method for logic programs is discussed based on the use of hash or associative memories for the binding environment and the database. The method is extension of that employed in H-Prolog system. Applications of the method are discussed both for serial depth-first evaluation and for heuristic (best-first) concurrent evaluation. In the heuristic evaluation, the processes share the common memories for the environments and the database. The serial heuristic evaluation is implemented to examine the usefulness of the method. Systems employing this method require no garbage collection cycle for the working memories and the databases, and can dynamically distinguish local variables from global variables to economize the memory usage.

1 INTRODUCTION

In this paper we present a general evaluation method for logic programs based on the use of either hash technique or associative (or content-addressable: CA) memories. The method is concerned with the design both of environments to represent the states of variables and of the database which contains the clauses.

In addition to nondeterminism as in other languages for artificial intelligence, e.g. PLANNER and CONNIVER, the following unique features of logic program execution make the environments more complicated.

(1) Some variables may have been defined in an earlier stage before being substituted.

(2) The memory space occupied by some variables (local variables) can be reclaimed after deterministic application of a clause.

Furthermore, we require multiple environments for heuristic evaluation, i.e. evaluation by heuristic (or best-first) search, including serial heuristic evaluation as well as OR-parallel or AND-parallel evaluation. A classification of the logic program evaluations is shown in Figure 1.

For the serial depth-first evaluation, most systems employ multiple stacks to store working data [Bruynooghe 1982, Mellish 1981, Warren 1977]. On the other hand, the stack is not a basic mean to realize multiple environments for the heuristic and/or parallel evaluation. It is not difficult for systems using lists as the main working data to realize the multiple environments by "association lists." The association lists, however, require

serial inefficient accesses and garbage collection to reclaim the garbage list cells.

A simple method to realize the multiple environments is to copy an environment at each new branch of the evaluation process. A drawback of this method is its inefficiency in memory usage and computation time due to copy operations. Another drawback is that the common variables cannot be used as a mean for communication between the AND-parallel concurrent processes as described in [Clark and Gregory 1981, Shapiro 1983].

Our method is extension of that employed in H-Prolog system [Nakamura 1983] we are developing. The H-Prolog system uses two hash memories, one of which is the working storage for the bindings based on structure sharing [Boyer and Moore 1972]. The other hash memory stores a part of data in the database for efficient comparison and access of the data: the clauses are represented by Lisp-like lists and every sublist without variables is stored in a hash memory as a "monocopy list" introduced by Goto [Goto 1974].

The hash memory also contains the indices of the clause heads for efficient selection of applicable clauses to goals.

2 A GENERAL MODEL OF LOGIC PROGRAM EVALUATION

Informally, an evaluation is a process to derive the empty goal list from an initial goal list (i.e., a question) and a program by linear input resolution [Kowalski 1979]. A program is a sequence of clauses of the form either

A.

or A :- $B_1,\ldots,B_n$.,

where A and B's are predicate terms. A goal list is a list of predicate terms, and represents a clause of negative literals.

2.1 Resolution

For a goal list L and a set (called an environment) E of bindings, let L:E represent the instance of L which is obtained by substituting its variables according to E. By resolution (or reduction) for a goal list L = $(G_1,\ldots,G_n)$ on a program P, a new goal list

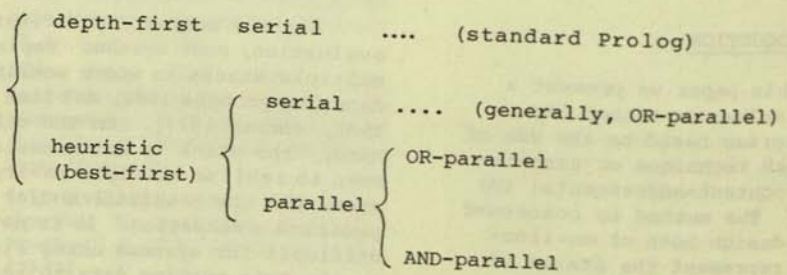


Figure 1. A Classification of Logic Program Evaluation

$$(G_1,\ldots,G_{i-1},A_1,\ldots,A_m,\ G_{i+1},\ldots,G_n):E$$

called a resolvent of L in P is derived, if unification of a goal G_i in L succeeds with a head of a clause $G :- A_1,\ldots,A_m$ in P and generates a set E of bindings. (If the clause is a unit clause, the subsequence $A_1,\ldots,A_n$ is regarded as empty.) Note that the variables in the clause should be renamed before each unification.

2.2 Join Operation for AND-Parallel Evaluation

Two or more goal lists are said to be AND-branching, if they are derived from a goal list L by the resolutions applied to different goals in L. It is necessary for AND-parallel evaluation to rejoin the AND-branching goal lists and generate a new goal list, if they are "consistent". More formally, two AND-branching goal lists

$$(G_1,\ldots,G_{i-1},A_1,\ldots,A_m,\ G_{i+1},\ldots,G_n):E$$

and

$$(G_1,\ldots,G_{j-1},B_1,\ldots,B_k,\ G_{j+1},\ldots,G_n):F,$$

$(i < j)$, can be joined into the list

$$(G_1,\ldots,G_{i-1},A_1,\ldots,A_m,\ G_{i+1},\ldots,G_{j-1},B_1,\ldots,B_k,\ G_{j+1},\ldots,G_n):E\cup F,$$

if both E and F do not contain bindings for a common variable with non-unifiable value terms. This operation can easily be extended to the operation for more than two AND-branching goal lists. Because the number of sets of AND-branching goal lists may be large, it is essential in the AND-parallel evaluation to assign a precedence to the goal lists and to allocate the operations to the processors.

2.3 Computation Graphs

We represent an evaluation of an initial goal list (a question) L_0 by a directed graph called a computation graph such that:

(1) Every node has a label, which is a goal list. The label of the root is L_0.

(2) If two or more edges enter a node, they represent the join operation. Otherwise, an edge represents a unification.

A node which has no leaving edge is called a terminal. A terminal is a success node, if its label is an empty list, or a failure node, otherwise. If an evaluation has no join operation, its computation graph is a tree. In this case each path from the root to a terminal represents an evaluation sequence.

Example Computation graphs for evaluations of finding common elements in two lists are shown in Figure 2 and 3. The program is for a definition of membership relation:

```
m(X,[X|_]).

m(X,[Y|L]) :- m(X,L).
```

3 CONTEXTS AND BINDINGS

In this section we discuss the working data based on the structure sharing.

3.1 Contexts

Our method uses values called contexts: the initial goal list has an initial context, and a unique context is generated at the

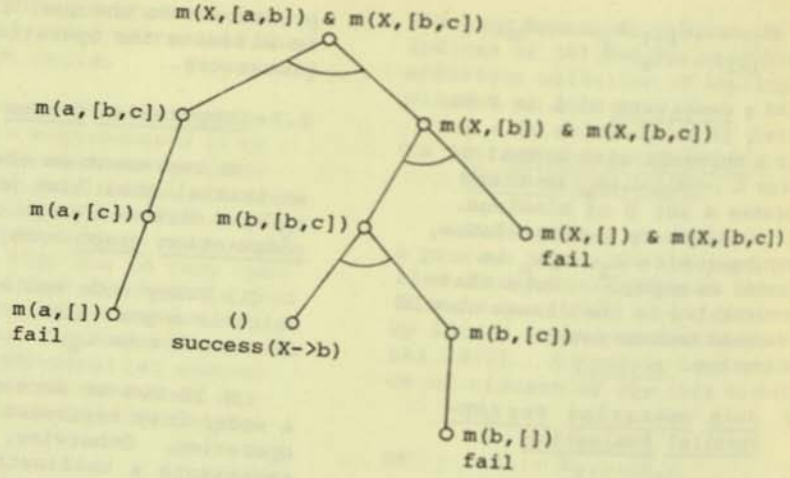


Figure 2. A Computation Graph for a Serial or OR-Parallel Evaluation of Finding Common Elements

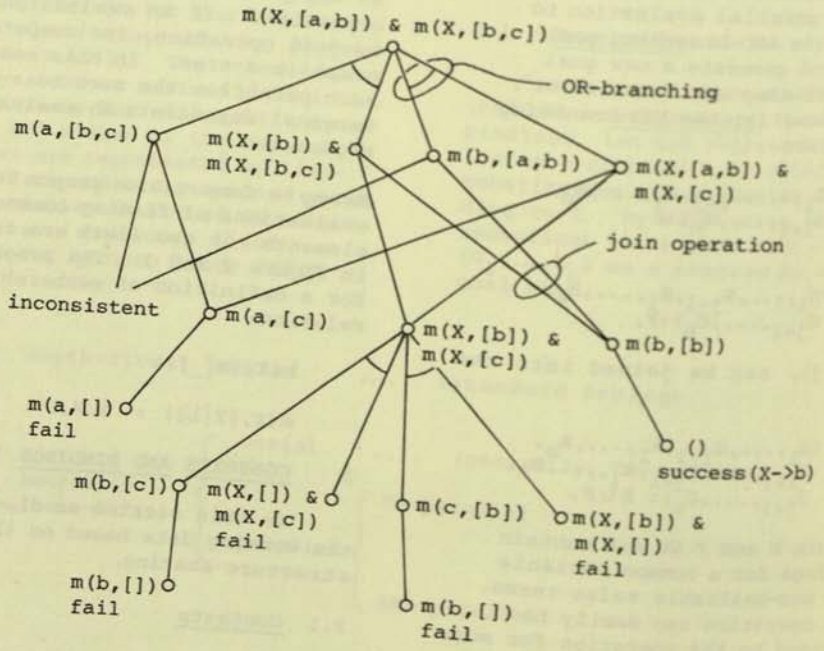


Figure 3. A Computation Graph for an AND-Parallel Evaluation of Finding Common Elements

beginning of each resolution. A context is used as a label not only of a resolution but also of the clause whose application begins with the resolution. Every variable in the clause is referred to with the context of the clause.

Let (s,t) denote the resolution of a goal with the context s and the clause with the context t, which is generated by this resolution. An application of a clause $A :- B_1,\ldots,B_m$ with a context t to a goal with the context s consists of the resolutions

$$(s,t),(t,i_1),(t,i_2),\ldots,(t,i_m).$$

The application is deterministic, if there is no other candidate clause in all the resolutions except (s,t) of the application.

The contexts are partially ordered: we write $i \leq j$, if and only if either $i = j$ or the resolution labelled i is followed by the resolution labelled j in a path of the computation graph. For any context c, we have $0 \leq c$, where 0 is the initial context.

3.2 Bindings

We represent a binding by

$$V_i \rightarrow_k t_j,$$

where i, j, and k are contexts with $i \leq k$ and $j \leq k$. This means that by a resolution labelled k a variable V with context i is instantiated to a source subterm (or a source sublist) t whose variables have a context j. The V_i represents a renamed variable, and the t_j an instance of t by structure sharing. Practically, V and t are the pointers to the variable and the source term, respectively.

3.3 Storage for the Contexts and Bindings

We assign one of the four states in Table 1 to each context. The states change as shown in Figure 4. The state R is assigned to a context when all the evaluation paths following the goal list with this context are found to have the failure terminals. The state N changes to D when the cut operator is encountered in a serial application of a clause.

The system holds the state information in an array or a hash or CA memory called the context table. In the case the hash or CA memory is used, it only contains context-state pairs with the state either D, N, or T and deletes the pair when the state changes to R.

Every binding $V_i \rightarrow_k t_j$ is stored in the hash or CA memory and accessed by its keys V and i. The system considers the variable V_i to be uninstantiated, if there is no binding for V and i in the memory or the context k in its binding is in the state R. If the system employs the CA memory and can access the binding by the context k, it can delete the binding when the application labelled k fails. The system employing the hash memory can detect the unused binding in the hashing or rehashing process and reuse its place for storing a new binding.

3.4 Links

In either serial or OR-parallel systems in which the left-most goals in goal lists are evaluated first, a goal list can be represented by a set of source sublists of goals and linkage information, as the instances of terms by source terms and the bindings. Suppose that a clause

G :- $B_1,\ldots,B_m$ with a context i is applied to G_1 in a goal list $(G_1,\ldots,G_n)$ with a context j. We represent the linkage information by the data of the form

$$()_i \ \rightarrow_i \ (G_2,\ldots,G_n)_j,$$

where $(G_2,\ldots,G_n)$ denotes the pointer to the sublist. We call these data the links.

4 LOCAL AND GLOBAL VARIABLES

Suppose that a binding $V_i \rightarrow_k t_j$ with $i \leq j$ and $i \neq j$ has been generated. We say that a variable with a context is global, if it occurs in t_j or its binding is used to construct the instance of t_j. The variable is local if it is not global. (Note that this definition is different from those in [Warren 1978, and Mellish 1980], in which local and global variables are determined statically.)

After deterministic application of a clause with a context j is terminated, i.e. the state T has been assigned to the context j, the system does not refer to the bindings for the local variables with the context j. Therefore

state of a context	application of the clause
D	deterministic, in progress
N	nondeterministic
T	deterministic, terminated
R	proved to be failure

Table 1. States of the Contexts

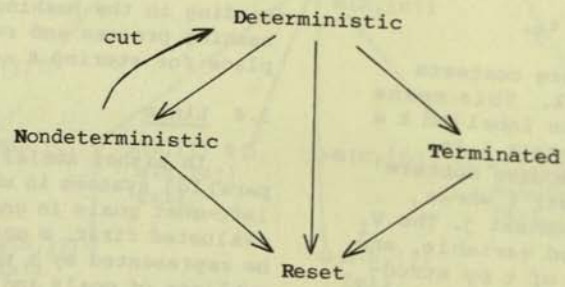


Figure 4. Possible State Changes

the places of these bindings in the memory can be reused to store another working data for economy of the memory, as those of bindings with the reset contexts.

To separate the local variables from the global variables, we attach one-bit information to each binding for indicating whether its variable is local or global. Because variables in a term which is instantiated to a global variable later are also global, uninstantiated global variables are also required to have this information. Therefore, we store special "bindings" of the form

$$V_i \rightarrow_k \underline{\text{uninstantiated}}.$$

for every uninstantiated global variable V_i.

The links are treated as the bindings for local variables.

5 SERIAL DEPTH-FIRST EVALUATION

Most Prolog systems employ serial depth-first search to find successful evaluation sequences. These systems backtrack to the ancestor node whenever they find that a goal list is failure. The state R (reset) is assigned to a context c, when the effect of application of the clause with context c is deleted in the backtracking. The system can store the bindings and the links in a hash memory, and refer to a binding or a link by its keys.

In the depth-first evaluation, the ordering of contexts is necessary only to determine whether a variable is local or global. We can simply assign the integer n to an n-th context for the ordering, since the system only tests the order for two contexts in an evaluation sequence.

6 HEURISTIC EVALUATION

Our model of the heuristic evaluation is illustrated in Figure 5. The processes or processors share the common associative memories for the environments, the database, and the context table, to execute the resolution and the join operations. The control unit maintains the set of goal lists to be evaluated, and allocates the operations to the processors.

In the heuristic, either serial or parallel evaluation, the system may store more than one binding (or link, in the case of serial or OR-parallel evaluation) with the same keys. A binding or link is valid only in the evaluation paths leaving the resolution which generates it. More formally, a binding

$$V_i \rightarrow_k t_j$$

is valid in a resolution with a label h, if and only if $k \leq h$ (This relation is illustrated in Figure 6). Therefore, we need some efficient method to test the relation $k \leq h$ and to select the valid binding or link for given X, i, and k.

A method to determine the partial order is to assign binary vectors K and H called "position vectors" to k and h, respectively, for any contexts k and h such that $k \leq h$, if and only if $K \supset H$, where the operator $\supset$ is bitwise implication. Figure 7 illustrates an assignment of the position vectors to the contexts in an AND-parallel computation graph.

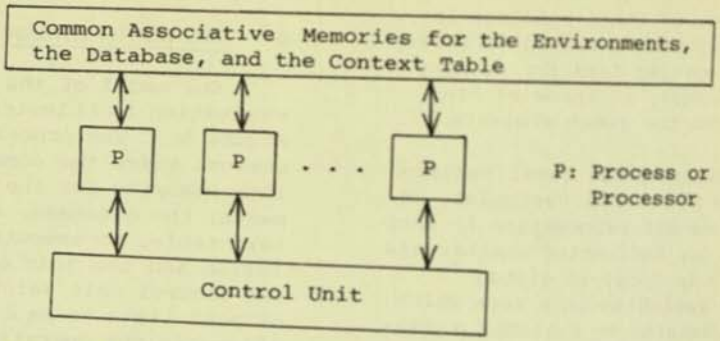


Figure 5. A Model of the Concurrent Evaluation

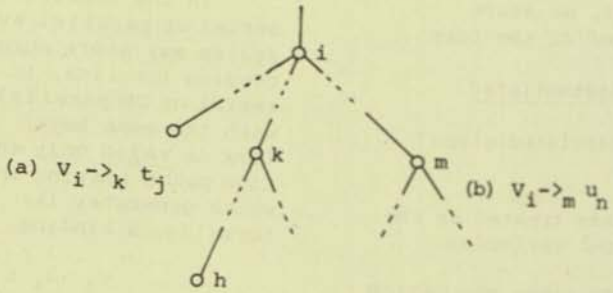


Figure 6. An Illustration of Relation such that the Binding (a) is Valid at the Context h

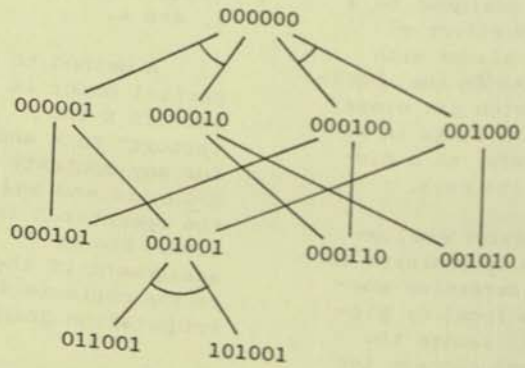


Figure 7. An Assignment of the Position Vectors to an AND-OR-parallel computation graph

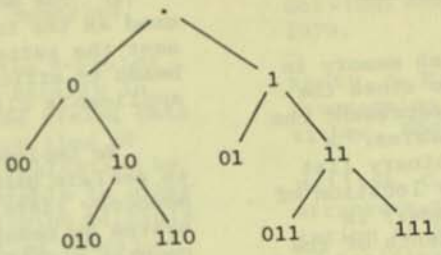


Figure 8. An Assignment of the Binary Strings to an OR-parallel computation graph

In serial heuristic evaluation and OR-parallel evaluation, we can employ another method. For any contexts k and h, we assign binary strings K and H to k and h, respectively, such that $k \leq h$ if and only if K is a rightmost substring of H. An assignment of the binary strings to an OR-parallel computation graph is shown in Figure 8.

If we can use an appropriate associative memory, the valid binding or link $X_i \rightarrow_k Y_j$ is determined for X, i, and h in a single, or small number of, operations by using the above method. On the other hand, the system using a hash memory requires to search the binding with $k \leq h$ for a set of bindings or links with the same keys X and i.

7 IMPLEMENTATION

The serial depth-first evaluation method is implemented in the H-Prolog interpreter [Nakamura 1983]. The H-Prolog system has been installed on several machines including large-scale and micro computers. Some advantages of our method compared to systems employing the multiple stacks are:

(1) The system written in the C language implicitly uses only one stack. This makes the program considerably simple.

(2) Local variables are dynamically distinguished from global variables. Hence, no garbage collection cycle is required for the memory storing binding information.

(3) Tail recursion (last call) optimization [Warren 1980, and Bruynooghe 1982] is easily implemented.

The serial heuristic evaluation method is implemented in C and H-Prolog to examine the usefulness of the method. The system maintains a queue of goal lists with their contexts to be evaluated. Since each goal list is composed of the source goal lists by means of the links, the actual elements in the queue are the pairs of pointers to the first source goal lists and their contexts. The goal lists in the queue are sorted according to an estimation of the goal lists, which determines the order of evaluation. We employ the binary-string method to determine the partial order of the contexts.

8 THE HASH MEMORY FOR THE DATABASE

We use another hash memory in the H-Prolog system to store the monocopy lists which represent the subterms without variables. A monocopy list is the binary list structure in which the location of each cell (or atom header) is determined by the contents of the cell (or the print name of the atom), and two pointers in a cell link to monocopy lists. Figure 9 illustrates the data structure of terms in the H-Prolog. Some advantages of this methods are:

(1) In the unification, equality of two subterms without a variable can be determined in a single step.

(2) The system can efficiently store a large quantity of data, provided that they can be represented as lists in which identical sublists occur frequently, e.g. parsed natural language sentences.

(3) We can easily add evaluation capability of Lisp-like functions and subprograms to the system.

(4) The monocopy lists can be used as the indices which represent the patterns of the clause heads to efficiently select the applicable clauses to goals.

We implemented the capability to reclaim garbage cells of the monocopy lists in the H-Prolog system by means of the reference counter method [Knuth 1968]. This garbage collection method is suitable for our system because used cells can be detected in the hashing or re-hashing processes as in the working memory. The system returns the garbage cells in the heap to the list of free cells whenever they change to be unused. Therefore the H-Prolog system requires no garbage collection cycle for its database as well as the working storage.

9 CONCLUDING REMARKS

We have described memory managements for the environments and the database in logic program evaluation systems based on the use of hash or content-addressable memories. It is clarified that we can realize a simple and efficient system for heuristic concurrent

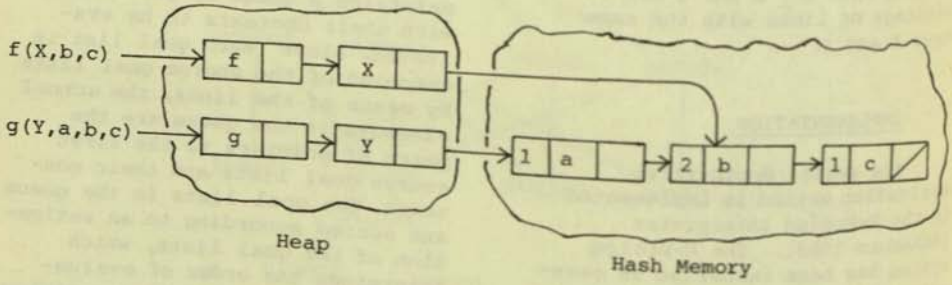


Figure 9. Data Structure of Terms in the Database
(The numbers in the cells are the reference counters.)

evaluation by the use of the content-addressable memories.

Efficiency of the H-Prolog system is discussed briefly in [Nakamura 1983]. Some timing data show that computation time of serial depth-first evaluation by the system is comparable to other Prolog systems employing multiple stacks. More detailed descriptions of the parallel execution method and efficiency of our methods will be appear in subsequent reports.

ACKNOWLEDGEMENT

The author would like to thank Professor Donald Michie of the Machine Intelligence Research Unit of the Edinburgh University, where he began this work. He acknowledges Isamu Shioya, and Masayuki Shimoji for helpful discussions and their assistance in the implementation.

REFERENCES

Boyer, R.S. and Moore, J.S. The sharing of structure in theorem proving programs, in Machine Intelligence 7 (eds. Melzer, B. and Michie, D.), Edinburgh University Press 1972.

Bruynooghe, M. The memory management of Prolog Implementation, in Logic Programming (eds. Clark, K. L. and Tärnlund, S. A.), Academic Press 1982.

Clark, K. L. and Gregory, S. Relational Language for Parallel Programming, Research Report DOC81/16, Imperial College 1981.

Goto, E. Monocopy and associative algorithms in extended LISP, Technical Report of Information Science Laboratory, University of Tokyo 1974.

Kowalski, R. A. Logic for Problem Solving, Elsevier North Holland 1979.

Knuth, D. E. The Art of Computer Programming 2, Fundamental Algorithms, Addison-Wesley 1968.

Mellish, C. S. An alternative to structure sharing in the Implementation of Prolog Programs, Dept. of Artificial Intelligence Research Report No.150, University of Edinburgh 1981.

Nakamura, K. Associative Evaluation of PROLOG Programs, Intelligent System research Group Memo TDU-ISRG-83-04, Tokyo Denki University 1983, also to be appear in Implementation of PROLOG (ed. Campbell, C. A.), Ellis Horwood.

Nakamura, K. H-Prolog (Version 1.0) Reference Manual, Intelligent System Research Group Memo TDU-ISRG-83-05, Tokyo Denki University 1983.

Robinson, J. A. A machine oriented logic based on resolution principle, Jour. ACM 12, pp.23-44 1965.

Shapiro, E. Y. A subset of Concurrent Prolog and Its Interpreter, Technical Report TR-003, ICOT 1983.

Warren, D. H. D. Implementing PROLOG - Compiling Predicate Logic Programs Vol.1, Dept. of Artificial Intelligence Research Report No. 39, University of Edinburgh 1977.

Warren, D. H. D. An improved Prolog implementation which optimises tail recursion, Proceedings of Logic Programming Workshop, pp. 1-11 1980.

A UNIFICATION ALGORITHM FOR CONCURRENT PROLOG

Jacob Levy
Department of Applied Mathematics
The Weizmann Institute of Science
Rehovot 76100, Israel

Extended Abstract

ABSTRACT

A new unification algorithm for Concurrent Prolog is presented. A previous implementation, written in Prolog, was extremely inefficient and incorrect in a central aspect. It did not solve the problems associated with true or-parallelism. The algorithm discussed in this paper forms the core of a new implementation of Concurrent Prolog in *C*, currently under development.

1 INTRODUCTION

Concurrent Prolog (Shapiro 1983), a variant of Prolog, supports concurrent programming and parallel execution. It combines a dataflow-like synchronization mechanism, guarded-command indeterminancy and a commitment method similar to nested transactions to create a powerful tool for parallel programming.

This paper reports our work on the analysis of unification in Concurrent Prolog. Previous descriptions of the language relied on unification being performed by the underlying Prolog system. These efforts did not allow correct implementation of or-parallelism (Shapiro 1983, p. 48). Therefore essential problems facing the implementation of unification were not addressed. The current paper presents these problems and proposes solutions.

The unification algorithm proposed here is intended to be executed on a uniprocessor. However, we conjecture that it can be extended to perform correctly in a multiprocessor system.

Section 2 describes Concurrent Prolog's syntax and defines its computation model. Section 3 discusses some of the special characteristics of a unification algorithm for Concurrent Prolog. Section 4 presents an algorithm that implements the required properties as described in Sections 2 and 3. In section 5 a discussion of the algorithm is presented.

2 SYNTAX AND COMPUTATION MODEL

2.1 Syntax

The syntax of Concurrent Prolog is the same as that of Prolog (Clocksin and Mellish 1981), with the addition of two new constructs, *read-only* annotation of variables, e.g.

$X?$, and the *commit* operator '|'. Both control the order in which a computation is performed and which clauses can be used.

A Concurrent Prolog program is a finite set of universally quantified guarded clauses of the general form

$$H :- G_1, G_2, \ldots, G_n \mid B_1, B_2, \ldots, B_m$$

$$m, n \geq 0.$$

H is the clause's *head*. The G's are the *guard*, and the B's are the elements of the clause's *body*. H, the B's and G's are atomic formulae, possibly containing variables, as in Prolog.

2.2 The Computation Model

At all times, there is a *current goal*

$$Q_1, Q_2, \ldots, Q_k \quad k \geq 0$$

that contains all individual goals that must still be solved. Initially, the current goal contains the conjunction entered by the user. In each cycle, the model chooses a goal Q_i, $k \geq i \geq 1$, and works on it.

Given a goal a, all clauses in the program that are potentially unifiable with the goal are selected (see Figure 1). The unification of the goal with the head of each clause is attempted.

If the goal a contains variables that may be instantiated through unification with the head of a clause or by the computation of the guard

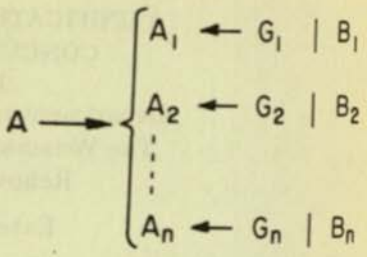


Figure 1

of a clause, copies of these variables are created. The local copies are then used instead of the variables appearing in the goal.

There are three possible outcomes of unification:

1. It may *fail*, in which case this clause is rejected.
2. Unification may *succeed*, and then the guard of the clause is created and associated with the goal (see Figure 2).
3. Unification may *suspend* because instantiation of a read-only occurrence of a variable to a nonvariable is attempted. The unification is delayed until the value of the variable becomes known, at which time it is retried.

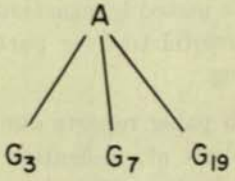


Figure 2

If unification causes a variable to unify with an occurrence of another variable marked read-only, all occurrences of the first variable be-

come read-only. Later attempts to instantiate this variable to a nonvariable must suspend.

When the computation of a guard terminates, the clause in which this guard appears, tries to commit. Two actions then take place. First, the computations of guards of all other clauses associated with the goal are discarded. Then the local copies of all variables made for this clause are unified with their counterparts in the goal. If the value of the local copy of a variable is not unifiable with the value of the corresponding variable in the goal, the commitment fails. In this way, if commitment succeeds, the values computed by the guard are exported to the variables in the goal and the goal becomes committed to these values.

If the unification of the local copies of variables with those in the goal requires instantiation of a read-only occurrence of a variable to a nonvariable, a suspension occurs.

Only one clause may attempt to commit for a given goal. If the commitment fails, other clauses are not allowed to attempt commitment of this goal at a later time. If commitment fails the computation of the goal also fails. This causes the system in which the goal appears to fail too, and the computations of the other goals in the guard are discarded. If commitment succeeds, the goal succeeds and is replaced in the guard by the body of the clause that committed.

A guard terminates successfully when all its subgoals succeed and are reduced to empty bodies. When some goal fails, the whole guard fails and is discarded.

A goal may thus fail for three reasons. It may be that the head of no clause in the program is unifiable with the goal. Another possibility is that all guards of unifiable clauses fail, and thus no clause commits. It is also possible that commitment fails because the value of the local copy of a variable and the value of the corresponding variable in the goal are not unifiable.

There are two possible outcomes from a computation: *success* and *failure*. The computation *halts* when the current goal is empty, and if the outcome is *success* the instantiated variables of the original goal are considered to be the output.

The occurrence of an infinite computation or a deadlock causes the current goal never to become empty. In this case the action of the model is not well defined.

3 CHARACTERISTICS OF UNIFICATION IN CONCURRENT PROLOG

3.1 Independence of Computations

In Prolog different paths to a solution are tried sequentially, and backtracking maintains the consistency of current instantiations for each path that is tried. In Concurrent Prolog it is not possible to rely on backtracking in this manner, be-

cause the guards of all clauses that were unifiable with a given goal are computed in parallel. The computation of each clause is independent of the computations of all other clauses and should not be affected by the instantiations they make. Therefore, local copies of all variables that appear in the goal must be made for each guard. Computation of the guard may then freely instantiate these local copies, without affecting the computation of other guards.

When a commitment occurs, the values of the local copies are unified with the values of the variables in the goal. This may fail if some brother of this goal instantiated these variables to values that are incompatible with those computed in the guard.

This scheme causes problems when a term that appears in the goal has some variables in it. Example 1 shows one type of difficulty.

Example 1

Goal – g(f(I)), I variable
Clause – g(A) :– ℓ(A), ... | ...
Clause – ℓ(f(3)).

The problem is caused by the fact that the unification of $f(I)$ with A does not ordinarily require that a new local copy of I be allocated. Therefore, I will be instantiated to 3 when the clause $\ell(f(3))$ commits, instead of when the clause $g(A)$:– ... commits. Thus I is instantiated too soon, with the effect that interference with other computations is possible.

A different kind of difficulty occurs precisely because the computations of all guards are independent, and so variables appearing in the goal are not instantiated to values from the guard until a commitment occurs. Example 2 presents a case where this causes a problem.

Example 2

Goal – a(X,X), X variable
Clause – a(b,C) :– g(C), ... | ...

The variable X should not be instantiated to b before commitment occurs, but C should be instantiated. However, in a naive implementation, because of the desire to keep guard computations independent, C would not be instantiated to b before commitment. This may cause problems if the result of $g(C)$ depends on the value of C.

3.2 Read-Only Variables

In Section 2 it was mentioned that instantiation of a variable to an occurrence of another variable marked read-only causes all occurrences of the first variable to become read-only also. Therefore, further attempts to instantiate it to a non-variable must suspend.

The following example illustrates this point:–

Example 3

Goal – g(X?), X variable
Clause – g(Y) :– Y = 34, ... | ... .

The goal $Y = 34$ must suspend until X becomes instantiated. At that

time the unification of Y with 34 is retried. This may fail, if the value of X is not 34.

Using a read-only annotation on an occurrence of a variable in a goal has the effect that the value is propagated into the guard as soon as it is needed by some computation and is available. For variables without read-only annotation the propagation is made, and the consistency check of the local and global copies is delayed until a commitment occurs.

Using a read-only annotation in the head of a clause is only effective if the variable will be instantiated to another variable in the goal. If the unification causes its instantiation to a nonvariable, a suspension will occur.

3.3 Propagation of Values

Consider a guard system

$$p(X), r(X) \; X \text{ variable.}$$

If p commits and instantiates X to some value, there are two possibilities. Either the computations of all guards under r are informed immediately of the value X was instantiated to, or the propagation is delayed until some clause commits for r. In the first scheme, it is possible that some guards will fail immediately if the value of X is not unifiable with their local copy. However, in the second scheme, such guards may reduce successfully and reach commitment. The original definition of Concurrent Prolog (Shapiro 1983) does not specify which of the two methods should be used.

4 ALGORITHM

4.1 Main Characteristics

The following points form the core of the algorithm.

(1) Suspension. The handling of suspension of unification consists of first undoing all instantiations done so far in this call and then saving the suspended process in a special queue allocated for each variable. When the variable becomes instantiated the processes in this special queue are restored to the current goal. Thus they are reactivated after the value of the variable becomes known.

(2) Demand Driven Copying. The algorithm is intended for a structure-copying implementation. To minimize copying so that only the needed information is copied, the notion of *demand driven copying* is introduced. Whenever a structure is copied, only the top-level functor is copied, and its arguments are initialized by special references to the original term called *get*-pointers. Only when a process requests access to an object referenced through a *get*-pointer, copying actually takes place. Thus, only the information that is actually accessed is copied. In the current presentation, an implicit assumption is that clauses are copied entirely whenever they are used in a com-

putation. It is thus not necessary explicitly to allocate local copies of variables since each clause has its own copies. Demand copying can also be applied to the copying of clauses and copying can be combined with the actual unification.

(3) Propagation by Request. The original definition of Concurrent Prolog does not specify how values of variables should be propagated into the computation tree. The current algorithm chooses the simpler scheme, that of propagating values only to those processes suspended on variables, i.e. propagation by *request*. The values of other variables are propagated only when commitment occurs.

4.2 Dereferencing

When one variable becomes unified with another, a chain of variables is formed. The unified variables are chained together with a *ref*-pointer. When the value of some object is desired, these chains must be traversed to retrieve the final value at the end of the chain.

The final value of a chain can be one of two types: a variable or a nonvariable. The possible combinations of entities that can appear in a chain are shown in Figure 3. The final results of dereferencing are summarized in Table 1. As can be seen, these results depend on the order of appearance of the different entities in the chain as well as on the final value of the chain. In case 3 of Figure 3 dereferencing stops as soon as a *get*-pointer appears. In case 4 a read-only annotation occurs before any *get*-pointers and so dereferencing continues through them. Dereferencing can be described as a finite-state machine, as is shown in Table 2.One or two results are returned, in $R1$ and $R2$. $R2$ is used for enqueing of suspended processes.

(1) – – – – (VAR or TERM)
(2) – – RO – – (VAR or TERM)
(3) – – – – GET
(4) – RO – – GET–{RO,GET or –}
– (VAR or TERM)

'–' normal reference
'RO' read-only annotation
'GET' get-pointer.

Figure 3

End of Chain		Case 1	Case 2	Case 3	Case 4
Variable	1	Variable	Last RO Variable	1st GET	Last RO Variable
	2				
Structure	1	Structure	Structure	1st GET	1st GET
	2				

Table 1: Results of dereferencing (cases as in Figure 3).

4.3 Demand Copying

The algorithm of demand copying is shown by the following two Pascal-like functions. Note that the exit code returned by dereferencing is used to determine the action.

```
function copy(Thing: ^ CP_Object):
  ^ CP_Object;
var ExitCode: Codes; R1, R2:
  ^ CP_Object;
begin
  deref(Thing,R1,R2,ExitCode);
  case(ExitCode) of
  VAR:
    R1 := REF(Thing);
    Thing := VAR(_);
    return Thing;
  RO:
    R1 := GET(R2);
    return RO(R1);
  TERM: return copyterm(R1);
  GET: return copy(copy(R1));
end;

function copyterm(Term):
  ^ CP_Object;
var Tmp: ^ CP_Object; i: integer;
begin
  Tmp := allocate(arity(Term));
  for i := 1 ... arity(Term) do
  arg(i,Tmp) := GET(arg(i,Term));
  functor(Tmp) := functor(Term);
  return Tmp;
end;
```

REF, VAR, GET and RO are procedures that return a reference, a new variable, a *get*-pointer and read-

State	Registers R1 : R2	REF	GET	RO	VAR	TERM
q0	0 : 0	pass -> q0	R1 := r ^ GET	R1 := r -> q1	R1 := r ^ VAR	R1 := r ^ TERM
q1	¬ 0:0	pass -> q1	R2 := r pass -> q2	R1 :=r pass -> q1	R2 := r ^ RO	R1 := r ^ TERM
q2	¬ 0 :¬ 0	pass -> q2	pass -> q2	R1 := r pass -> q2	R2 := r ^ RO	R1 := R2 ^ GET

'r' – The current element.
-> q1 – Transition to state 'q1'.
^ VAR – Return with exit_code 'VAR'.
pass – Make 'r' point to next element in chain.

Results as per Table 1 are returned in R1 and R2.

Table 2: Finite-state machine for dereferencing objects.

only term respectively. The references created by REF in *copy* are saved and undone at the end of each unification.

4.4 Unification

The unification algorithm is summarized in Table 3. *Unify_terms* recursively unifies two structures or atoms. *Suspend* saves the current goal in the queue of the variable, undoes all instantiations, and terminates this unification. FAIL undoes all instantiations in this call and terminates this unification with failure. The procedure *u* recursively calls unification on the result of copy.

Unification has two modes. The normal mode is used when the head of a clause is unified with a goal. All references created by REF during unification are saved and undone when it ends, regardless of the result. When commitment occurs, unification of the local copies of variables is attempted with their counterparts in the goal; then, unification does not undo the references created by REF but leaves them intact. Also, *get*-pointers created by *unify* in normal mode and not yet copied through, are modified to references.

Since the arguments to *unify* are dereferenced before the unification actually begins, the exit code of *deref* can be used to proceed directly to the correct entry in Table 3.

5 DISCUSSION

Several properties of the algorithm are worth noting:-

(1) The algorithm cleanly implements the requirement that substitutions in the goal be postponed until a commitment occurs. Global variables that become instantiated are propagated to a goal upon request, whereas variables instantiated in a guard do not propagate their value to outside it prior to a commitment.

(2) The algorithm provides an elegant solution to the problem of maintaining or-parallel environments in the form of demand driven copying. Only information that is used is copied.

Head T1 \ Goal T2	VAR	RO(V2)	TERM	GET
VAR	T1 := GET(T2)			
RO(V1)	T2 := REF(T1)	Suspend(V2)	FAIL	u(T1,copy(T2))
TERM	T2 := REF(T1)	Suspend(V2)	unify_term	u(T1,copy(T2))
GET	T2 := REF(T1)	u(copy(T1),T2)		

Table 3: Unification.

(3) The implementation of unification can make use of information gathered in the process of dereferencing to choose its actions. Thus excessive tests at runtime can be avoided. All cases of Table 3 except *unify_terms* can be compiled to simple macro instructions similar to those proposed by Warren (1980, 1983).

ACKNOWLEDGMENTS

This work was carried out jointly with my advisor, Ehud Shapiro. The author wishes to thank Nir Friedman for many useful discussions and criticism. This work was supported by IBM Poughkeepsie, Data Systems Division.

REFERENCES

Shapiro, E.Y. A Subset of Concurrent Prolog and Its Interpreter. Technical Report TR-003. ICOT – Institute for New Generation Computer Technology, Tokio, February 1983.

Clocksin, W.F. and Mellish, C.S. Programming in Prolog. Springer-Verlag, 1981.

Warren, D.H. An Improved Prolog Implementation Which Optimizes Tail Recursion. DAI Research Paper, No. 141, Dept. of Artificial Intelligence, University of Edinburgh, July 1980.

Warren, D.H. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, October 1983.

A MEMORY MANAGEMENT MACHINE FOR PROLOG INTERPRETERS

Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro
I.R.I.S.A. – I.N.R.I.A.
Campus Universitaire de Beaulieu
Avenue du général Leclerc
35042 Rennes – Cedex
France

ABSTRACT

Indeterminism is one of the original aspects of logic programming. On a mono-processor, indeterminism is solved using a simple sheduling policy called "backtracking". Such policy implies that there is some means, usually called "trailing information", for retrieving the state of suspended goal statements. The use of the trail is well known for backtracking. Unfortunatly its use in the design of garbage collection mecanisms has been ignored so far. We suggest to take advantage of it to get the exact state of suspended goal statements while performing the marking process of a garbage collector. A more complete garbage collection is therefore obtained. The intermediate machine which is proposed in this paper takes these concepts into account. A parallel implementation of the garbage collector is also discussed.

Introduction

Memory allocation is an important problem in the design of Prolog interpreters. Warren (Warren 1977) has done a first original step towards a solution for this problem, by rising static criterium for the differentiation of the life time of variables. Mellish (Mellish 1980) and Bruynooghe (Bruynooghe 1980) have also presented solutions including a copy method which uses more dynamic criterium. Warren (Warren 1980) has studdied a dynamic method which optimizes tail recursion. Finally, we owe to Bruynooghe (Bruynooghe 1982) the recommendation to start the marking from the active goal statements. This was the first step towards a specific, well fitted, Prolog oriented, garbage collector.

The marking algorithm which is presented in this paper starts the marking from the active goal statements, but we advocate that the trailing information should be used to get the exact state of non current active goal statements. Not using the trail for such a marking leads to keep unnecessary information (Bekkers et al. 1983).

In the following, a Prolog interpreter is introduced as a mean for extracting minimum concepts for the design of a well suited intermediate machine, supporting Prolog interpreters and handling memory management. A realisation of this machine is then given in terms of two parallel processes, one for resolution, the other for garbage collection.

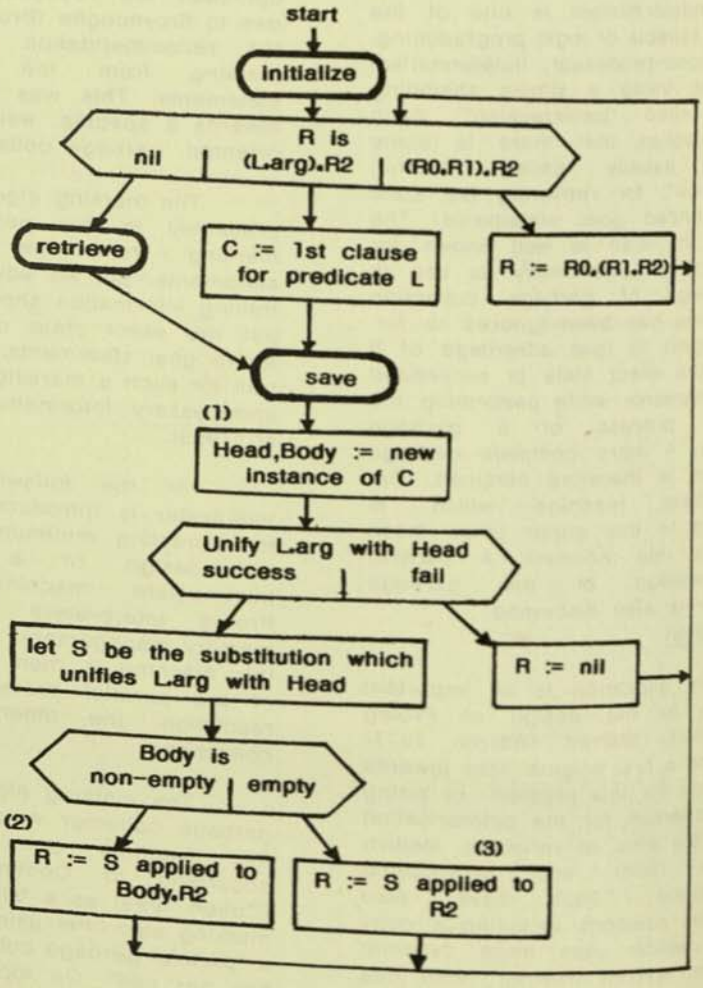
The marking algorithm and the garbage collector which are used in this machine can be classified according to Cohen classification (Cohen 1999) as a "single-sized cells marking algorithm using a stack" and a "parallel garbage collector using two bits per cell". On top of that, since

we have no need for compaction, the algorithm does not perform information moves (Bakers 1978).

1. A PROLOG interpreter

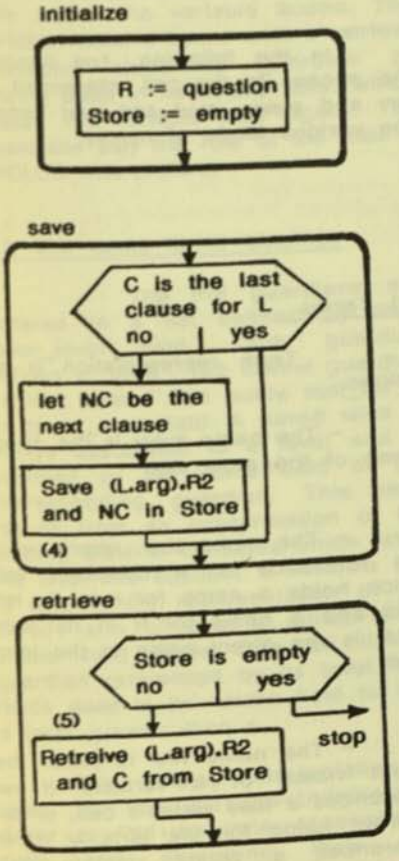
The PROLOG interpreter, detailed in figure 1, is a straighforward transcription of the resolution principle. Goal statements are objectively manipulated as binary terms.

A goal statement is either *nil*, in which case it is the empty goal statement, or a construct *(R1.R2)*, where *R1* is a non-empty goal statement and *R2* is a goal statement. A non-empty goal statement is either a goal *(L.arg)*, where *L* is an atom identifying a predicate and *arg* is a binary term standing as the argument of the goal, or a construct *(R11.R12)*, where *R11* and *R12* are non-empty goal statements.



1 rewriting the current goal statement

Figure 1. shows the cyclic rewriting of the current goal statement. The need to "create variables" appears in (1). "terms construction", "sub-terms selection" and "variables substitution" are needed in (2) and (3). Figure 2 shows the management of indeterminism using backtracking. The need to "save" goal statements appears in (4) and the need to "retrieve" them appears in (5). These basic operations have been implemented on an intermediate machine used by the PROLOG interpreter, hencefoward called "the user".



2 management of indeterminism

2. The intermediate machine

The intermediate machine has a state composed of a "top-level", which allows the user to signify sub-terms of the current goal statement by means of "names", and a "store", which is an ordered set of terms, each of them being a saved goal statement. The machine keeps the correspondance between names and terms.

2.1. Commands

The user invokes commands, by means of which it exchanges names with the machine. The user initially knows no names but those of atoms. It gets other names by means of commands. The commands connected with memory management are

construct(*ln*,*rn*:**name**) :**name**
create_variable :**name**
substitute(*vn*,*tn*:**name**)
save(*n*:**name**)
retrieve :**name**
reduce(*n*:**name**).

The result of **construct** is a name for a term the left and right sub-terms of which are respectively signified by the names *ln* and *rn*. Like in LISP, the machine has some commands, not described here, to obtain the components of binary terms.

The remaining commands are related to PROLOG's concepts of variable and indeterminism. The result of **create_variable** is a name for a new variable. The **substitute** command makes names in the top-level signify more instanciated terms: after the command, every name signifies the term signified before, where the variable signified by *vn* is replaced by the term signified by *tn*. The **save**

command pushes the term signified by *n* in the store, the top-level remaining unaffected. The result of **retrieve** is a name for the term popped from the store. This significance becomes the top-level.

After the **reduce** command, the top-level is reduced to the significance of *n*. Invoking this command allows the machine to collect every cell which does not participate to the representation of either the term signified by *n* or the saved terms. The collected cells are rendered available for future use. Instructing the machine about the user's accesses is a more flexible technique than using a fixed number of access registers known by the machine: the user can store temporarily in its own memory an unlimited number of names which are accesses in the top-level, for example during unification.

2.2. Implementation

The implementation of the machine supports two processes: the "user process" which invokes commands, and an internal "garbage-collector process" which works in parallel at the recovery of the memory resources no longer useful for the representation.

3. The representation

The basic data types are **reference**, **data** and **name**. The machine has a memory which is a collection of cells, each one addressed by a **reference**. Every cell can be considered under any of the formats **construct**, **variable** or **guardian**.

reference = {**null_ref**}∪{1..maxref}
data = {0..**maxdata**}

name = structure
| **indicator** :{**a**,**c**,**v**}
| **information** : **reference** or data
construct = structure
| **left** :**name**
| **right** :**name**
variable = structure
| **status** :{**free**,**uncertain**}
| **level** :**reference**
| **binding** :**name**
guardian = structure
| **nature** :{**live**,**dead**}
| **lower_level** :**reference**
| **name** :**name**

In the following, *↑ref* denotes the access to the cell referenced by *ref*, and **a**:*data*, **c**:*ref* and **v**:*ref* denote the various kinds of names.

3.1 Terms

Term representation is as follows:

- The name **a**:*data* is the direct name of the atom *data*.

- The name **c**:*ref*, where *ref* is the reference of a construct cell which holds a name for *lt* in its left field and a name for *rt* in its right field, is the direct name of the term (*lt*.*rt*).

- The name **v**:*ref* is either the direct name of a variable if *ref* references a free variable cell, or an indirect name for the term *bt* if *ref* references a bound variable cell which holds a name for *bt* in its

binding field. The free or bound state of a variable cell is determined differently, whether it is considered from the top-level or from a saved term.

3.2. The top-level bindings

Considered from the top-level, a variable cell is free either if its status field holds **free** or if the nature field of the guardian referenced by its level field holds **dead**. When a cell is allocated for a new variable, its status field is initialized **free**. After variable substitution, the variable's status field holds **uncertain** and its level field holds the reference of a live guardian. This makes the variable bound. The variable remains bound until a **retrieve** command alters its guardian by storing **dead** in its nature field, which makes the variable free again. The guardians play the role of the trail in PROLOG interpreters.

3.3. The saved terms bindings

The live guardians are ordered in a list defined by their lower_level field. Each guardian defines a "level". The lowest guardian defines level 0 and holds **null_ref** in its lower_level field. A saved term is always associated to a level and is signified by the name field of the corresponding guardian. This term results from an interpretation of the binding state of variables which takes the level into account: a variable cell is free in the representation of the saved term of level *k* either if its status field holds **free** or if the guardian referenced by its level field holds **dead** in its nature field or has a level greater than *k*.

This binding interpretation is used by the garbage-collector in order to find accurate accesses to cells and determine their real usefullness. Most of existing systems, derived from those implemented for LISP, do not take into account the new dimension introduced by indeterminism and interpret the bindings of variables independently of the level of the observed goal statement. This amounts to consider over-instanciated goal statements, and leads to retain more cells than necessary.

4. The user process

The **user_level** register contains the reference of the guardian at the highest level. Each command has a corresponding procedure. The **search_direct_name** procedure goes through variables bindings to yield the direct name equivalent to a given name. This induces shorter chains of bound variables and increases the opportunities of loss of access to variables.

The **substitute** command subscripts the binding of a variable with the reference of the guardian at the highest level. The effect is to validate this binding in the top-level, as long as the guardian remains alive. Therefore, the top-level is always associated with the highest level. The **save** command saves a given term by storing its name in the guardian at the highest level and creates a higher level guardian which will be used as subscript for the next bindings in the top-level. The **retrieve** command causes the death of the guardian at the highest level. This undoes the variable bindings according to the top-level interpretation. Then the command restores at the top-level the guardian right beneath, and returns the name of the associated saved term. The **reduce** command triggers the garbage-collector, supplying it with a name which constitutes the only access kept by the user. The head of the guardians list is automaticaly passed to the collector, and defines the root of the accesses due to saved terms.

5. The garbage-collector process

The garbage-collector works cyclicaly. Each cycle is called a "batch" and consists of a marking phase followed by a collecting phase.

The **marking_level** register references the guardian which defines the level currently under marking. The **marking_name** register contains the name which is the root access to the cells representing the saved term to be walked through.

```
user_level :reference

procedure create_variable :name
| ref_v:=cell_allocation
| ↑ref_v.status:=free
| result v,ref_v
procedure construct(left_n,right_n:name) :name
| ref_cons:=cell_allocation
| ↑ref_cons.left:=search_direct_name(left_n)
| ↑ref_cons.right:=search_direct_name(right_n)
| result c,ref_cons
procedure substitute(nv,nt:name)
| ref_v:=search_direct_name(nv).information
| ↑ref_v.binding:=search_direct_name(nt)
| bind_variable(ref_v)
procedure save(n:name)
| ↑user_level.name:=search_direct_name(n)
| ref_g:=cell_allocation
| ↑ref_g.lower_level:=user_level
| ↑ref_g.nature:=live
| user_level:=ref_g
procedure retrieve :name
| re_adjust_level
| unbind_variables
| user_level:=↑user_level.lower_level
| result ↑user_level.name
procedure reduce(n:name)
| start_batch(n)

procedure search_direct_name(n:name) :name
| nn:=n
| loop while nn.indicator=v
| | | and test_absolute_binding(nn.information)=bound
| | nn:=↑(nn.information).binding
| result nn
```

```
marking_name :name ;   marking_level :reference

process garbage_collector
| block marking_phase
| | loop while marking_level≠null_ref
| | | mark_term(marking_name); down_one_level
| block collecting_phase
| | ref_cel:=0
| | loop while ref_cel<maxref
| | | ref_cel:=ref_cel+1
| | | make_cell_available(ref_cel)
| wait_next_batch
procedure mark_term(n:name)
| ref:=n.information
| if n.indicator≠a and test_mark(ref)=unmarked then
| | case n.indicator
| | c then mark_term(↑ref.left); mark_term(↑ref.right)
| | v then if test_relative_binding(ref)=bound then mark_term(↑ref.binding)
| | cell_marking(ref)

procedure bind_variable(ref_v:reference) exclusion bindings
| ↑ref_v.level:=user_level; ↑ref_v.status:=uncertain
procedure unbind_variables exclusion bindings
| ↑user_level.nature:=dead
procedure test_absolute_binding(ref_v:reference) :{free,bound} exclusion bindings
| if ↑ref_v.status=free or ↑(↑ref_v.level).nature=dead then result free
| else result bound
procedure test_relative_binding(ref_v:reference) :{free,bound} exclusion bindings
| if ↑ref_v.status=free then result free
| else
| | ref_g:=↑ref_v.level
| | case test_mark(ref_g)
| | marked then result free
| | unmarked then
| | | if ↑ref_g.nature=dead then ↑ref_v.status:=free; result free
| | | else result bound



procedure down_one_level exclusion position
| decrement_level
procedure re_adjust_level exclusion position
| if marking_level=user_level then
| | suspend garbage_collector
| | decrement_level
| | restart garbage_collector
procedure decrement_level
| cell_marking(marking_level)
| marking_level:=↑marking_level.lower_level
| if marking_level≠null_ref then
| | marking_name:=↑marking_level.name
```

5.1. Marking levels in decreasing order

The marking phase proceeds level by level in decreasing order. This yields the important property that the marking visits each usefull cell exactly once, when marking the highest level which has access to this cell. This can be informaly justified as follows. Let *i* and *j* be two levels, with *i*<*j*. Any **construct** cell yields the same immediate accesses for these two levels. Any **variable** cell yields at level *i* an immediate access either empty or identical to the access it yields at level *j*, because a variable bound at a given level remains identically bound for any higher level. Therefore, the set of cells accessed via a given cell at level *i* is included in the set accessed via this cell at level *j*.

This property is illustrated on figure 3. The sets of cells S1 and S0 respectively represent the terms of levels 1 and 0, kept in association with the guardians C11 and C13. While marking level 0, it is not necessary to go through the cells beyond cell C5, already visited at level 1, because all the cells this would lead to are already marked.

Not taking the levels into account would lead to wrongly consider cells C3 and C4 useful.

After a level has been marked, the procedure **down_one_level** marks the corresponding guardian. Thus, comparing the level of the bindings with the level under marking simply amounts to test the allocation mark of the guardians.

```
current_batch :{0,1}
garbage_collector_idle :{true,false}
procedure wait_next_batch exclusion batches
| garbage_collector_idle:=true; suspend garbage_collector
procedure start_batch(n:name) exclusion batches
| if garbage_collector_idle then
| | current_batch:=(current_batch+1) mod 2
| | marking_level:=user_level; marking_name:=n
| | garbage_collector_idle:=false; restart garbage_collector

procedure cell_marking(ref:reference)
| status_alloc[ref]:=current_batch
procedure test_mark(ref:reference) :{marked,unmarked}
| if status_alloc[ref]=current_batch then result marked
| else result unmarked

available_cells :reference
status_alloc :array 1..maxref of {0,1,available}
procedure cell_allocation :reference exclusion allocation
| if available_cells=null_ref then wait cell_available
| cell_marking(available_cells); result available_cells
| available_cells:=↑available_cells.next
procedure make_cell_available(ref:reference) exclusion allocation
| if status_alloc[ref]=(current_batch+1)mod 2 then
| | status_alloc[ref]:=available
| | ↑ref.next:=available_cells; available_cells:=ref
| | signal cell_available
```

5.2 Incidence of parallelism on marking

The execution of **retrieve** by the user process may lead to the logical destruction of the level currently under marking. In this case, the marking is aborted and the garbage-collector starts to mark the level below. Aborting a marking is possible if cells already marked by the collector do not give access to unmarked cell. To insure this condition, the cells get marked from the leaves to the root.

The parallel execution of the user process does not affect the principle of considering encountered marked cells as leaves. Indeed, cells allocated since the beginning of the current batch lead to cells which were allocated either during this batch, in which case they were marked at allocation time, or before this batch, in which case they can be accessed from the root given to the garbage-collector at batch start, and will be marked when encountered during the walk from this root.

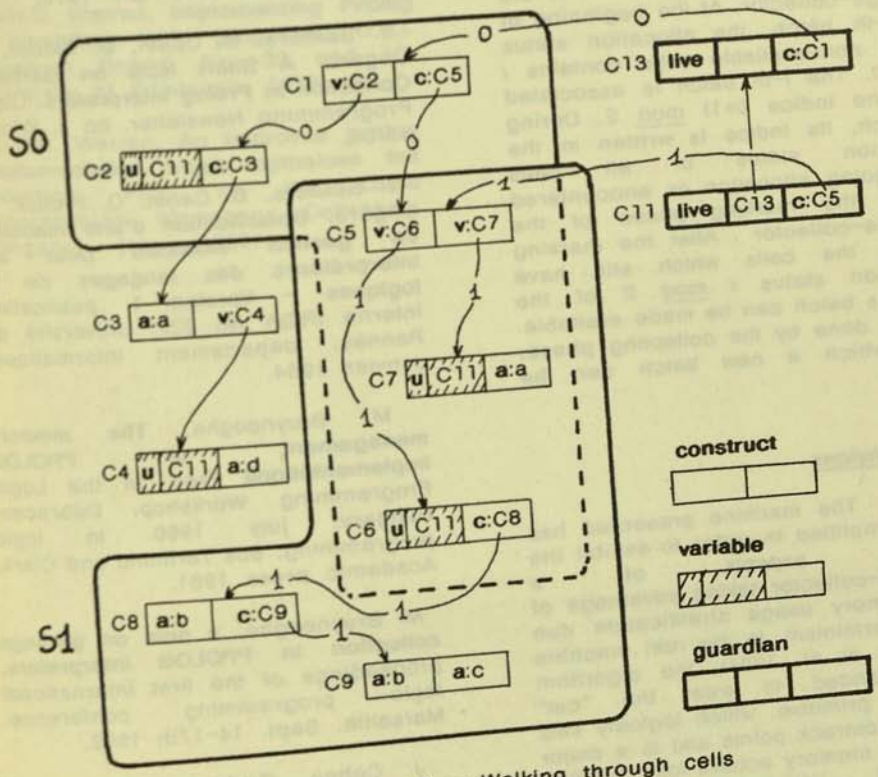


figure 3 - Walking through cells

6. Cell allocation, batches

The array **alloc_status** keeps a current allocation status for every cell. There are three status possibilities: **available**, qualifying an available cell, and the two batch indices 0 and 1, qualifying an allocated cell. The **current_batch** register contains the indice of the current batch. This indice is used to mark the cells at the time they are allocated, or when they are encountered by the collector in its marking phase.

Batches

Batches are introduced in the memory management to cope for parallelism between the user and the garbage-collector. At the beginning of the *i*-th batch, the allocation status of all non-available cells contains *i* mod 2. The *i*-th batch is associated with the indice (*i*+1) mod 2. During a batch, its indice is written in the allocation status of all cells undergoing allocation or encountered during the marking phase of the garbage-collector . After the marking phase, the cells which still have allocation status *i* mod 2 of the previous batch can be made available. This is done by the collecting phase, after which a new batch can be started.

7. Extensions

The machine presented has been simplified in order to exhibit the original aspects of a garbage-collector taking advantage of the memory usage stratification due to indeterminism. In the real machine (Bekkers et al. 1984), the algorithm are extended to treat the "cut" PROLOG primitive, which logicaly kills saved backtrack points and is a major cause of memory access loss, leading to full valorisation of the garbage-collector.

A simulator of the intermediate machine and a PROLOG interpreter using it are currently under development. A hardware realisation with two processors is under study. One of them will be microprogrammed to support the major part of the intermediate machine.

REFERENCES

H.G. Baker, **List Processing in Real Time on a Serial Computer** Communications of the ACM, Vol. 21 No. 4, p. 280-294, April 1978.

Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro, **A Short Note on Garbage Collection in Prolog Interpreters**, Logic Programming Newsletter, no 5, Winter 83/84.

Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro, **Spécification d'une machine de gestion mémoire pour les interpréteurs des langages de la logiques - Version 1**, publication interne IRISA No 222, Université de Rennes, département informatique, Janvier 1984.

M. Bruynooghe, **The memory management of PROLOG implementations** Proc. of the Logic Programming Workshop, Debrecen, Hungary, july 1980. in logic programming, eds Tarnlund and Clark, Academic press 1981.

M. Bruynooghe, **a note on garbage collection in PROLOG interpreters**, proceedings of the first international logic programming conference, Marseille, Sept. 14-17th 1982.

J. Cohen, **Garbage Collection of Linked Data Structures**, ACM Computing Surveys, Volume 13 Number 3 1981, p. 341-367.

E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens, **On-the-Fly Garbage Collection: An Exercise in Cooperation**, Communications of the ACM, Vol. 21 No. 11 966-975, Nov. 1978.

C.S. Mellish, **An alternative to structure-sharing in the implementation of a PROLOG interpreter**, Proc. of the Logic Programming Workshop, Debrecen, Hungary, july 1980. Logic Programming, K.L. Clarck and S.A. Taernlund (eds.), Academic Press, 1981, also in: Research Paper no 150, Departement of Artificial Intelligence - University of Edinburgh.

D.H.D. Warren, **Implementing Prolog - compiling logic programs**, D.A.I. Research Report, No. 39 and 40, University of Edinburgh, 1977.

D.H.D. Warren, **An improved prolog implementation which optimises tail recurtion**, Proc. of the Logic Programming Workshop, Debrecen, Hungary, july 1980.

AUTHOR INDEX

Abramson, H. 77
Bekkers, Y. 345
Berger Sabbatel, G. 207
Bosco, P.G. 219
Broda, K. 301
Canet, B. 345
Chikayama, T. 1, 89
Crammond, J.A. 183
Dahl, V. 77
Dang, W. 207
Dershowitz, N. 315
van Emden, M.H. 35
Eriksson, L-H. 101
Furukawa, K. 89
Giandonato, G. 219
Giovanetti, E. 219
Goguen, J. 115
Gregory, S. 301
Hattori, T. 1
Hirakawa, H. 89
Janeselli, J.C. 207
Johansson, A-L 243
Josephson, N-A 315
Kacsuk, P. 195
Kale, L.V. 171
Kurokawa, T. 1
Lassez, J-L. 263
Levy, J. 335
Lindstrom, G. 159
Littleford, A. 289
Lloyd, J.W. 35
Maher, M.H. 263
McCord, M. 65
Meseguer, J. 115
Miller, C.D.F. 183
Nakamura, K. 323
Nguyen, G.T. 207
Nilsson, M. 13
Pique, J.F. 23
Plaisted, D.A. 151
Rayner, M. 101
Ridoux, O. 345
Sakai, K. 1
Sato, T. 127
Shoham, Y. 277
Sintzoff, M. 139
Štěpánková, O. 53
Štěpánek, P. 53
Sterling, L. 231
Sun, H. 253
Szöts, M. 41
Takagi, S. 1
Tamaki, H. 127
Tsuji, J. 1
Uchida, S. 1
Ungaro, L. 345
Wang, L. 253
Warren, D.S. 171
Wolfram, D.A. 263
Yokoi, T. 1

Program Committee

K.A. Bowen, Syracuse University, USA
M. Bruynooghe, Leuven University, Belgium
K. Fuchi, ICOT, Japan
H. Gallaire, Laboratories de Marcoussis, France
K.M. Kahn, Uppsala University, Sweden
P. Köves, SZKI, Hungary
F.G. McCabe, Imperial College, UK
F. Pereira, SRI, USA
L.M. Pereira, Universidade Nova de Lisboa, Portugal
J.A. Robinson, Syracuse University, USA
E. Shapiro, Weizmann Institute, Israel
S.-Å. Tärnlund, Uppsala University, Sweden, Chairman
M. van Caneghem, University of Marseille, France