AB43.4.3 ABSTRACTOd Thoughts.

Hendrik Boom, Mathematisch Centrum, 2e Boerhaavestraat, Amsterdam

Abstract

Some design aspects of a language intended for verified programs are discussed in relation to transformational programming. The data type system of such a language can be intimately related to intuitionist logic.

Computing Reviews categories: 4.20, 4.29, 5.24.
AMS-MOS classification: 68A-30, 68A-40, 02C15.

Key words and phrases: Abstracto, transformational programming, verification, data types, intuitionism, propositional calculus.

1. FOREWORD

In the last few years, Working Group 2.1 has started investigating the transformational approach to programming, in which the programming process is seen as starting with some correct program which expresses some computation in a clear language. By applying semantics-preserving transformations, this program is then converted to a form which can be efficiently implemented on a conventional machine. This is not the same as the more traditional stepwise refinement process, which operates by continually implementing previously unimplemented features. The transformation approach can better be viewed as analysis, translation, and optimization than as refinement. Throughout the transformation process, the program is expressed in an at present undefined language called "Abstracto". Programs in Abstracto need not be directly executable; it is envisaged that Abstracto will contain implementable and unimplementable features, as well as specifications and other useful things. The aim is to support the programming process from first conception of algorithm to final expression as executable code (in "Concreto"). There has also been talk of a language "Transformo" in which to guide the transformation process. The discussions on these points are still extremely vague.

This is a working paper presented at the WG 2.1 meeting at Jabłonna. It is deliberately speculative, and was written to suggest ideas, not to present conclusions of completed research. This may account for a certain half-bakedness in the style of presentation. Minor changes have been made to make it more accessible to the uninitiated reader.

2. OLYMPO

This paper investigates what language can be at the top of the hierarchy. Let us call it Olympo, to make it a highest-level member of the -o family of languages. Olympo should be designed principally to make verification easy, and efficiency be hanged. Perhaps efficiency can be transformed in after, but then we are speaking of Abstracto or even Concreto instead of Olympo.

The current state of the art in formal verification techniques is at present very dependent on automatic theorem proving. The normal approach seems to be to probe a program with assertions much as an acupuncturist inserts strategic pins into a patient, to look at the reaction, and to pass the resulting verification conditions to an automatic verifier. The verification conditions tend to be extremely complicated, probably because attempts are made to let the assertion language mimic faithfully all the convolutions of operational semantics.

An additional source of complexity is that the present-day formal proof and assertions are variations of the n-th order predicate calculus for small n. Predicate calculus was not originally constructed for the practical verification of practical theorems. It was instead viewed as a minimal set of axioms sufficient for the formalization of mathematics, so that the necessary use of logic in mathematics can be precisely studied. Serious attempts to use predicate calculus to formalize mathematics involve massive use of abbreviation. This should be a warning. Present-day assertion languages should be compared to machine languages, and we should seriously attempt to raise their level. A concerted attack on this problem may yield the same kind of progress as we have achieved in high-level languages over the last thirty years.

3. INTEGRATED VERIFICATION

Since the advent of Structured Programming (fanfare on trumpets), there has been a growing suspicion that the correctness proof of a program should be constructed at the same time as its code. Much of the present work on verification ignore this point; in particular, it is ignored in any design for a compiler that accepts a program, generates object code, and also generates verification conditions to be checked (or not) by a separate verifier.

The most successful form of automatically verified assertions to date has been strong data typing. It is extremely rare for a programmer to be tempted to leave out the data types in a strongly-typed language because they involve too much work, or because he can see that the program is correct anyway. The data types are an integral part of the program, and not merely an add-on feature. The temptation to omit assertions is very strong with other mechanical verification schemes. These are clearly add-on techniques. We may conclude:

> The assertion mechanism of Olympo should be so well-integrated into the language that no one would think of leaving out an assertion in the hope of getting a program ready fast.

Each programming language feature should simultaneously generate object code and a proof. It may well be that there may be a variety of features with the same object code but different proof rules. Each would be used in a different situation, and be chosen to make a different application easy to prove. For example, consider the ordinary integral for-loop:

```
for i from 1 to n do S(i) od,
```

where S(i) is a parameterized statement. Various axioms may be convenient to verify programs involving such a loop:

(1) Axiom "for1"

```
For i in 1..n,
   Let V(i) be a set of variables,
   Let S(i) involve only variables in V(i)
   Let P(i) be a predicate.
   Let P(i) involve only variables in V(i)
   Let V(i) and V(j) be disjoint for i ≠ j
   Let {true} S(i) {P(i)}
Then
   {true} for i from 1 to n do S(i) od {FORALL i in 1..n: P(i)}
```

(2) Axiom "for2"

```
for i in 0..n, let P(i) be a set of predicates.
for i in 1..n, let {P(i-1)} S(i) {P(i)}.
Then {P(0)} for i from 1 to n do S(i) od {P(n)}
```

(3) etc.

It is true that axiom (1) can be derived from axiom (2). However, axiom (1) is much easier to use than axiom (2), and is often sufficient. If loops with different axioms were to have different syntax, verification might be simplified for common constructions such as:

```
   for i from 1 to n do A[i] := 0.0 od.
```

With the hint that axiom (1) suffices, this loop can be verified (postcondition FORALL i in 1..n A[i]=0.0) without the trouble of inventing an induction hypothesis.

There are probably no more than 10 to 30 commonly used loop structures. It would not be hard to include each of them in a programming language, perhaps as part of a standard prelude. Each of the above for-loop constructions can be considered as a procedure call. The procedure is the for-loop axiom, and its arguments are the pieces of program text "1", "n", "lambda i: S(i)", and the assertions that these pieces of program have specific properties. (Alternatively, these assertions may be considered to be included in the data types. More about this later.)

4. BOUND VARIABLES IN MODES

Consider a procedure which accepts an integer i and yields an array of size i. In Algol 68 we would be able to write its mode as PROC(INT)[]REAL. But why not PROC(INT i)[1:i]REAL? This involves a bound variable in the mode, but provides more information [3].

Bound variables within data types become more useful if one permits the parameters to be modes, and permits their modes to depend on previous parameters. The traditional example is:

```
   PROC sort = (MODE M, REF[]M arr, PROC(M,M)BOOL less) VOID:
      C sort the array 'arr' according to the ordering 'less' C.
```

Such formal mode parameters have been traditionally called "modals" in WG 2.1 [2].

5. DATA TYPES AS PROPOSITIONS

It is possible to use the conventional concept of data type (or mode) as extended above to encode the intuitionistic propositional and predicate calculi. Propositions are represented by data types (and not by values of some fixed data type).

A proposition P, when encoded as a data type P, can be considered as the data type of all its proofs. A value of mode P then represents a proof of P. The existence of a value of such a type is equivalent with the provability of the proposition. To make this work, it is of course necessary to abolish facilities like SKIP and NIL, which produce fake values of an arbitrary mode; otherwise everything would become provable. Nonterminating procedures must also be abolished.

The implication P => Q provides a means of converting proofs of P into proofs of Q. We therefore encode P => Q as PROC(P)Q. A proof of P => Q is a procedure which will turn any proof of P into a proof of Q.

We can now construct procedural proofs of tautologies such as P => ((P => Q) => Q). To avoid notational confusion below, we shall use the word PROC as in Algol 68 when we are constructing a mode, but shall write the word LAMB in front of every routine-text. P => ((P => Q) => Q) is represented by the mode PROC(P)PROC(PROC(P)Q)Q. To prove P => ((P => Q) =>

Q), we must construct a procedure of this mode, that is, a routine text starting

```
   LAMB(P a)PROC(PROC(P)Q)Q : ...
```

This accepts "a", a proof of P, and produces a proof of (P => Q) => Q. The "..." must be a routine text of mode PROC(PROC(P)Q)Q, so we may write

```
   LAMB(P a)PROC(PROC(P)Q)Q :
      LAMB(PROC(P)Q ab) Q : ...
```

But now it is easy to obtain an object of mode Q {i.e a proof of Q} by writing ab(a) {combining the proof ab and the proof a}, thus:

```
   LAMB(P a)PROC(PROC(P)Q)Q :
      LAMB(PROC(P)Q ab)Q : ab(a).
```

The call ab(a) corresponds to the use of modus ponens.

It is clear that Algol 68 routine texts are not the best vehicle for expressing proofs. Conventional proofs may even be a better notation for some Algol 68 programs. Some combination of the two may eventually turn out to be appropriate.

The conjunction P AND Q is represented by

```
   MODE P AND Q = STRUCT(P first, Q second).
```

That is, to construct a proof of P AND Q one must combine the separate proofs of P and Q. Furthermore,

```
   MODE P OR Q = UNION(P, Q),
   MODE FALSE = (PROC(MODE M) M),
   MODE NOT P = (P => FALSE).
```

With these definitions, it is possible to find procedures that prove all of Heyting's axioms [1] for the propositional calculus (see the appendix).

Typed universal quantification is straightforward:

```
   MODE FORALL i:T P(i) = PROC(T i) P(i),
```

using the parameterized procedure-yield modes we have already seen. Typed existential quantification can be done with parameterized unions, but we can do it with modals instead:

```
   MODE THEREEXISTS i:T  P(i) =

      PROC(MODE R, FORALL i:T (P(i) => R) ) R.
```

6. IMPLICATIONS FOR PROGRAMMING LANGUAGES

The above results strongly suggest that it is not necessary to separate the assertion language from the data type system, nor to separate the proof language from the procedural language. It may even be undesirable, since the possibilities that may be opened by the direct execution of proofs of theorems are still largely unexplored.

7. FEATURES IN AND OUT OF OLYMPO

Olympo must have facilities for the free construction of programs out of components. The components must be expressible with a high degree of abstraction in order to increase their applicability. Furthermore, there must be no facilities which could lead to lurking side effects, since these would greatly complicate verification.

Therefore:

NO

- side effects
- assignments,
- variables
- GO TOs
- explicit input/output.

These restrictions may well be too severe for anything resembling normal programming; they are introduced so that we can have an easier problem to solve before we start on a hard one. The solution to the easy one may show us how to avoid the hard one entirely.

MAYBE

- heuristic choice and backtracking.

YES

- procedures accepting multiple arguments and yielding multiple results.
- arbitrary typed parameters.
- lots of data types.
- procedures and programs as parameters.
- free algebra with induction law as a primitive data type constructor.
- well-ordering in some form (for termination)
- identity declarations
- promissory notes (see below).

A "promissory note" is a promise that in a later version of a program, some component will be provided that is now missing. The compiler can proceed with syntactic and semantic checks on the rest of the program. Proofs, code, types, and assertions may all be missing in this sense. It is clear that the program cannot be expected to run until essential parts are provided.

It should also be possible to specify "undefined" types and predicates. It may be too much work (or even impossible) to formally write down the complete definition of some predicate or proof. An undefined predicate can then be useful, provided that there is some mechanism for letting the programmer claim that it is satisfied. The compiler can then propagate the assertion throughout the program as part of a data type or otherwise, and can test whether it has been duly claimed whenever it is required to hold. There will then be two kinds of assertions:

- Announcements, where the compiler believes the programmer, and
- Assertions, where the compiler checks the programmer, possibly using the announcements or other means.

8. SUBTYPES

If assertions are to become part of the data types of objects, we are going to have to have convenient ways of adding and removing assertions from a data type. Removing assertions can be left to a coercion (this is similar to using a subtype when a type is required in the DOD's languages), but adding assertions will have to be done by a proof or as an implicit consequence of a run-time test (IF test THEN test true ELSE test false FI).

APPENDIX

Intuitionistic propositional calculus can be built from data types: procedural proofs of Heyting's axioms are listed below. Each entry in the following list is of the form "proposition --- proof".

```
P => (P AND P) ---
   LAMB(P a) STRUCT(P first, second): (a, a);

(P AND Q) => (Q AND P) ---
   LAMB(P AND Q ab) (Q AND P): (second OF ab, first OF ab)

(P => Q) => ((P AND R) => (Q AND R)) ---
   LAMB(P => Q ab) ((P AND R) => (R AND R)):
      LAMB(P AND R ac) (Q AND R) :
         (ab(first OF ac), second OF ac)

((P => Q) AND (Q => R)) => (P => R) ---
   LAMB((P => Q) AND (Q => R) abbc) (P => R) :
      LAMB(P a) R: (second OF abbc)((first OF abbc)(a))

Q => (P => Q) ---
   LAMB(Q b) (P => Q): LAMB(P a) Q: b

(P AND (P => Q)) => Q ---
   LAMB(STRUCT(P first, (P => Q) second) aab) Q :
      (second OF aab)(first OF aab)

P => (P OR Q) ---
   LAMB(P a)UNION(P, Q): a

(P OR Q) => (Q OR P) ---
   LAMB(UNION(P, Q) ab) UNION(Q, P) : ab
      # Algol 68 mode equivalencing does this one ! #

((P => R) AND (Q => R)) => ((P OR Q) => R) ---
   LAMB((P => R) AND (Q => R) abbc) ((P OR Q) => R) :
      LAMB(P OR Q ab) R :
         CASE ab IN
            (P a): (first OF acbc)(a),
            (Q b): (second OF acbc)(b)
         ESAC

NOT P => (P => Q) ---
   LAMB(P => (MODE M)M af)(P => Q) : LAMB(P a)Q : af(a)(Q)

((P => Q) AND (P => NOT Q)) => (NOT P) ---
   LAMB((P => Q) AND (P => NOT Q) abab) (P => FALSE) :
      LAMB(P a) FALSE :
         (second OF abab) (a) (first OF abab(a)).
```

REFERENCES

[1] Heyting A., Intuitionism, North Holland Publishing Company, 1951.
[2] Lindsey, C.H., Modals, Algol Bulletin AB37.4.3, July, 1974.
[3] Boom, H.J., Extended type checking, in New Directions in Algorithmic Languages, 1976, pp. 27-42, IRIA, also separately as Mathematical Centre report IW60/76, Amsterdam.