stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 76/77 JANUARI
(DEPARTMENT OF COMPUTER SCIENCE)

D. GRUNE

TOWARDS THE DESIGN OF A SUPER-LANGUAGE OF ALGOL 68 FOR THE STANDARD PRELUDE

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A15

ACM-Computing Reviews-categories: 4.12, 4.22

Towards the Design of a Super-language of ALGOL 68 for the Standard Prelude[*)]

by

Dick Grune

ABSTRACT

Some problems are examined which arise in the design of an Unabridged Machine-Independent Standard Prelude for ALGOL 68 in a form which can be handled by a compiler. Two of these problems, one concerned with the character set and the other with SIZETY declarations, are examined in depth and tentative solutions are given.

KEYWORDS & PHRASES: ALGOL 68, language design, standard prelude, portability.

[*)] This paper is not for review; it is meant for publication elsewhere.

1. Introduction.

Generally a program is allowed to use certain identifiers which have not been declared in the program, but are somehow known to the system. Examples include the trigonometric functions, square root, etc., and they come under different names like "intrinsic functions" or "standard externals".

Generally they are known to the user because their names appear in a list in the manual, with explanations; they are known to the compiler because the same list has been built into it, with references to run-time routines.

In the ALGOL 68 Report the situation is slightly different. The list and the explanations are both there, but are supplied as a sequence of declarations in a super-language of ALGOL 68; these declarations constitute the "Standard Prelude".

Now it would no doubt be possible to use the procedure outlined above and just incorporate all this information in the compiler, a technique which has worked well for FORTRAN, PL/I, etc.

For ALGOL 68, however, the approach has clear disadvantages, mainly caused by the fact that we want to provide the implementer with a machine-independent system. We are in the unfortunate position of not knowing what features will be available in his installation (e.g., whether or not he has a window symbol (I), or how many L bytes he is going to have). Consequently we would not know what to incorporate in the compiler. Even if he cares to tell us he would be in trouble if he changed his mind afterwards.

Moreover, the information about "standard" facilities is supplied in a more explicit and algorithmic form than in any other language, and it would be nice to utilize this by actually implementing the super-language of the Standard Prelude.

This approach would also simplify some problems with the transput part of the standard prelude. A standard prelude is an excellent way of providing large parts of the run-time system, which must otherwise be supplied through a different mechanism that might look conspicuously like a standard prelude.

The super-language of ALGOL 68 in which the standard prelude is written is described in RR 10.1.3 in terms of textual substitution. This textual substitution often cannot be performed in actual practice (long, etc.). And there are cases where the text of the prelude is correct, but highly impractical (definition of * on integers).

One could, in spite of these difficulties, formalize syntax and semantics of these substitutions in a fashion similar to the rest of the report, and then incorporate these features in the compiler. This would place a heavy burden on the compiler (and designer), but the text of the standard prelude could be used as it stands.

But it seems advantageous to slightly modify the syntax and semantics so as to ease implementation and aid portability, at a minimum of expense in flexibility.

We try to reach this goal by judicious application of a few tools:

- macro-preprocessing (to solve some problems of machine-dependency,
- small changes to the identification mechanism (for "long sin", etc.),
- a general escape mechanism for most of the other problems.

The resulting super-language should be easy to implement and allow easy changes and extensions to the standard prelude, both at the language support level and at the local level.

2. Selecting the Problems.

Among the many files that will eventually constitute the Mathematical Centre Machine-Independent ALGOL 68 Compiler, there will be a file containing in some form the Unabridged Machine Independent Standard Prelude, henceforth to be called the UMISP.

When supplying a UMISP we are immediately forced to consider the character set to distribute it in. The character set implied by the Report is not included in any standard character set. We could, however, define a suitable eight-bit extension of ASCII 128, and use it for distribution purposes. The recipient would then be confronted with the task of weeding from the UMISP the declarations of those operators that could never occur in his system.

To avoid endless discussion and implementation problems we decree that the UMISP be punched in the 60 character Hardware Representation Set as described by H. J. Boom and W. Hansen [1] (to be called 60RS), which every ALGOL 68 installation will have to have.

This UMISP in 60RS must, among other things, contain:

- a. all operator symbols which are not represented in 60RS,

- b. invisible declarations,
- c. an indeterminate number of copies of declarations and code (especially in the transput),
- d. an infinity of names (long long ... sin),
- e. a few things that for fundamental reasons cannot be written in ALGOL 68 (e.g. the "funny" generator in RR 1Ø.3.5.k),
- f. replacement code for text that is correct ALGOL 68, but unacceptable in a reasonable implementation (for example, the declarations of mode file, op (int, int) int over, ...),
- g. strings that cannot be punched in 6ØRS ("$_{10}$\Ee" in RR 1Ø.3.3.2.a).

In order to appreciate the difficulties, consider the plight of someone receiving our portable ALGOL 68 compiler. He receives one standard package (tape) of files, and expects to be able to produce from this, by a reasonable effort, an ALGOL 68 system adapted to his local circumstances. He has a (not too ample) superset of 6ØRS, routines for integer and real arithmetic, can plunder complex arithmetic from the FORTRAN run-time system, has nothing for long real, but has a student working on it. And our Standard Prelude mechanism should be flexible enough to work for him rather than against him.

We first made an inventory of the problems by reading the standard prelude and making notes of all features of which it was not obvious that they would easily fit into a UMISP. This resulted in nine pages of grim reading.

The problems can be grouped together in many ways, all rather unsatisfactory. One especially unattractive way is always open: to find an ad-hoc solution to each and every problem; but we hope to do better.

Being thus confronted with an unstructured mass of problems we decided to select problems of which we expect that their solution will create order in the chaos.

The selected problems are:
- the actual character set is unknown beforehand, and the necessity to change it may arise after the compiler has been installed,
- the number of distinct long's and short's is unknown beforehand, and will very likely change after the compiler has been installed.

3. The Character Set.

Once the compiler is installed it should be able to make full use of the local character set; it would be silly (and confusing) to have an ≤ symbol and not be able to use it. So the compiler must be able to read an ≤ and know what it is; it can only have obtained this knowledge by reading the standard prelude. But this standard prelude is in 60RS and does not contain ≤ . Moreover, the compiler is code-independent too, cannot have the representation of ≤ built-in, and would not even know to expect it when reading a defining operator.

So the first information the code-independent compiler lacks is knowledge about what constitutes a defining operator. A list of these should be supplied to the compiler, on a file in the local character code that we shall call the Representation File. Having read this file the compiler knows what is a defining operator, and it could exploit this knowledge in reading the UMISP, but for the fact that the UMISP is in 60RS, and does not contain ≤ directly.

We know, however, that the UMISP is, as the name says, unabridged and contains declarations for ≤ , ⌶ , etc., in some form. So we can apply a program (something like a macro-processor) which reads the Representation File and subsequently converts the embryonic declarations in the UMISP into full-fledged declarations in the local character code, at the same time deleting anything which cannot be expressed in the local code.

Such a converter can be very simple. Essentially it has only two tasks, to convert longhand operators into local shorthand, and to skip text conditionally. It could react to keywords starting with ./ and know character constant denotations starting with .= , the values of which it obtains from the Representation File.

The declaration of e.g. the window-symbol operator on bits could read (in the UMISP):

```
./ifce .=w op .=w = (int a, bits b) bool: (& of b) [a]; ./fi
```

where ´ifce´ means ´if character exists´. The converter would produce from this:

```
op ⌶ = (int a, bits b) bool: (& of b) [a];
```

It is clear that this example is pertinent only as far as the window symbol is concerned: all other problems in the original declaration are still present.

The standard prelude contains 19 different characters that are not represented in 60RS, viz., 16 operators, backslash, low ten and lower-case e for the input of reals. Each of these can be assigned a single letter and be known to the converter as .=<letter>.

This scheme solves the problem of the machine-dependent character set: the compiler first acquaints itself with the local character set by reading the Representation File, then creates a local version of the standard prelude, and through reading it acquaints itself with the locally available operators.

Of course this need not be the actual sequence of events upon compilation of each and every program. A separate converter could create the Local Standard Prelude, and the compiler could precompile this LSP into a set of tables in some internal form that would later serve to inform the compiler of the standard operators whenever translating an actual program.

4. The SIZETY problem.

When considering the form of a SIZETY operation definition in the UMISP several approaches come to mind, none of which work. All solutions fail because the user may write:

long long sin(leng leng 3.0)

in an environment in which ´real lengths´ equals 1; no technique which is confined to the prelude mechanism only can cope with this.

Some solutions seemed very attractive at first and it is useful to show here why they don´t work.

- Proposal: let the Representation File contain values for ´real lengths´, ´int shorths´, etc., and let the converter generate the appropriate declarations.
  Objection: programs like the one above cannot be handled.

- Proposal: introduce a genuine

  Lint = union(int, long int, ...)

  Objection: this would legalize forms like ´long sin(3.1)´ or ´long 3.1 + 3.1´ and have adverse effects on the run-time efficiency. The technique may prove usable in the transput section.

- Proposal: let the first scan of the compiler find out the maximum number of long´s used in the program under consideration. It can then generate all declarations that

could ever be used in this program. The information necessary for this generation could be provided by the standard prelude.
Objection: operator identification has not yet been done when the maximum number of long's is going to be determined. This makes cases like

```
leng if b then x else y fi
```

hard to handle (increase all SIZE counters by one?).

We are forced to make changes to the compiler itself, more specifically to the identification mechanism, which must be generalized to comprise SIZETY declarations. This immediately raises an important question. These changes will no doubt extend the power of the language considerably. Should this new facility be made available to the user? If so, we can stick to the exact form of the Report and allow declarations like

```
op * = (L compl a, L real b) L compl: a * L compl(b)
```

both in the standard prelude and in user programs. The answer to this question will strongly affect the details of the design of the extension.

At first sight the reasonable answer seems to be "yes". The user who is developing a matrix-handling package will certainly be grateful to us, and in general it is good practice to restrict system privileges to a minimum.

Upon closer inspection, however, some unpleasant phenomena come to light.

- Well-formedness.
  If the user is allowed to define his own L-modes, checking well-formedness is awkward and can depend on the number of long's in the application. Example:

```
mode u = union(int, long long int)
mode L yech = union(u, ref union(u, L int))
```

  Now L yech is well-formed for all numbers of long's except Ø and 2! The standard prelude itself does not contain such monstrosities.

- Equivalencing.
  New modes will be created during operator identification in this scheme. These modes can, in devious ways, be equivalent to other modes, and this equivalence may be essential for the identification of other operators. So mode equivalencing and operator identification must form a

single integrated block, a prospect we do not relish.

It can be objected that this situation will occur whatever we decide; a construction like

```
    leng if b then x else y fi
```

will give rise to new modes when the operator leng is identified. But the crux lies in the words "in devious ways". The user can (and will) concoct examples that need the full power of mode equivalencing, by using unions of L-modes. However, if the L-modes are restricted to those of the standard prelude, mode equivalencing is almost trivial and can easily be handled during operator identification. The hardest case is the lengthening and shortening of L compl.

- Generality.

  Once we give the user the possibility to declare modes like the L yech above, we are forced by the spirit of ALGOL 68 to allow modes that depend on two or more SIZETY parameters, e.g., L1 L2 yecchh. This might be useful, but it is a bit beyond the scope of this subject.

- Independence.

  The concept of 'independence of properties of declarations' as used in RR 7.1 becomes unclear. It is hard to decide whether or not the following two declarations should be dependent.

```
    (a) op www = (L int a) L int: a;
    (b) op www = (int a) int: -a;
```

  If (a) is visible when we try to identify the operator www in www 1, it should be identified, and likewise for (b); this means that (a) and (b) cannot co-exist in the same range and that they must be considered dependent.

  If, however, (a) is in an outer range and (b) in an inner range, and we try to identify www in www long 1, we find that (b) should not render (a) inaccessible, in other words, that they should be independent.

  The standard prelude itself does not raise this problem, since it does not contain declarations that are equally similar as (a) and (b).

These considerations force us to reject the idea of L-modes as a general feature. At the same time they indicate that the use of L-modes in the standard prelude is essentially simpler than the normal use of modes, and it would be nice to exploit this simplicity. Some minor simplifications have already been given under the headings "Well-formedness" (no check necessary), "Equivalencing" (trivial for standard prelude modes) and "Generality" (one SIZE parameter only), but the great bonus comes from analyzing the problem mentioned under "Independence".

The trouble with declarations (a) and (b) is that the modes of their operands are firmly related for some "values" of the L-parameter, in which case their properties are "not independent" in the sense of the Report. The standard prelude, of course, does not contain any pair of declarations that is dependent for some value of L . Thus there cannot be an applied occurrence of an operator with L-mode operands that would identify one declaration for one value of L and the other for another value of L . But this means we can afford to completely disregard the number of long's and short's when doing the identification. If we identify a declaration, then either it is the correct one, or there is no identification possible. We see that we can use the normal operator identification mechanism for the standard prelude as well, if we are prepared to do some additional checking.

Checking is required to catch cases like

```
    compl z;
    z +:= struct(real re, long real im) (0, long 1)
```

where the right hand side reduces to struct(real re, real im) upon reaching the standard prelude, and consequently the operator +:= on compl in RR 10.2.3.11.f is identified.

The problem resembles the "false" operator identification in

```
    int i; real x;
    if b then x else i fi +:= 3.0
```

in a compiler that uses operator identification by H-function as described by Hendrik Boom in [2] (which our compiler will). Here the representative mode of the left hand side is ref real and the operator +:= of RR 10.2.3.11.e is identified. A separate check is then necessary to find out that i cannot be coerced to ref real. Such a check can profitably be incorporated in the coercion mechanism. It can also catch falsely identified L-mode operators.

Thus the identification of standard prelude operators is extremely simple: when reaching the standard prelude discard all SIZETY information. The coercion process will then determine the value of L from one of the L-mode operands and check coercibility as usual. It can at the same time determine which of the different sources in the declaration applies in this case.

This suggests declarations of the form:

```
op ('l real, 'l int) 'l real + =
     [ <0 ]: source 1,
     [ =0 ]: source 2,
     [ =1 ]: source 3,
     [ >1 ]: (long long real a, long long int b)
              long long real: a + long long real(b)
```

The source for L > 1 is correct if in this implementation both 'int lengths' and 'real lengths' equal 2, since then there is no run-time difference between the data structures for long long real, long long long real, ... (and likewise for integers).

Note that the + in the last line can be identified without reference to the value of L, and has indeed been so identified during the translation of the standard prelude.

The UMISP will contain something like

```
op ('l real, 'l int) 'l real + =
     [ <0, =0, >0 ]: (real a, int b) real: a + real(b)
```

and the user can extend this as he implements more lengths and shorths.

Similar methods can be used for identifiers that start with "long". Again the "long"s are stripped when reaching the standard prelude and identification proceeds as usual. Again a check is necessary to prevent cases like "long sin(3.14)".

The indicated technique hinges on the fact that the standard prelude does not contain declarations which become dependent when SIZETY information is disregarded. Upon closer scrutiny it appears that in addition the standard prelude contains no declarations which become dependent if PREF information is disregarded. This implies that upon entering the standard prelude we could strip off all ref's and proc's as well. It is not clear whether this is an advantage. It does make the identification in the standard prelude simpler at the expense of making it different.

5. Acknowledgement.
I am very grateful to Lambert Meertens for his many ideas, his patient explanations of difficult points and his numerous counterexamples.

6. References.

[1] Boom, H. J., W. Hansen, The Report on the Standard Hardware Representation for ALGOL 68, ALGOL Bulletin 40.5, 1976; also available as IW 64/76, Mathematical Centre, Amsterdam, 1976.

[2] Boom, H. J., Note on Balancing in ALGOL 68, ALGOL Bulletin 36.4.1, 1973.