Towards the Design of a Super-language of ALGOL 68 for the Standard Prelude (excerpt)

by

Dick Grune
Mathematical Centre
Amsterdam
March 1977

Abstract.
The problems concerning SIZETY definitions in an Unabridged Machine-Independent Standard Prelude for ALGOL 68 are examined and tentative solutions are given.

Keywords.
ALGOL 68, language design, standard prelude, portability.

The SIZETY problem.

The design of the Mathematical Centre Machine-Independent ALGOL 68 Compiler calls for a text containing in some form the Unabridged Machine-Independent Standard Prelude. This text must include, among other items, definitions for operators and identifiers with modes involving SIZETY. When considering the form of such a SIZETY definition 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.

Some solutions seemed very attractive at first and it is useful to show here why they don't work.

- Proposal: feed values for 'real lengths', 'int shorths', etc. to the compiler and let it generate the appropriate declarations.
  Objection: programs like the one above cannot be handled.

- Proposal: introduce a genuine

  ```
  mode 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 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 l)
```

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 [1] (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. The value of L can be passed to the code generator for selecting the appropriate object code.

Acknowledgement.
I am very grateful to Lambert Meertens for his ideas and his numerous counterexamples.

References.

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