RA

stichting
mathematisch
centrum

MC

REKENAFDELING MR 135/72 AUGUST

L. MEERTENS

ON STATIC SCOPE CHECKING IN ALGOL 68

RA

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
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), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

0. Introduction

0.0. Some remarks concerning code optimization.

Consider the following particular-program:

```
(for k to maxint do
    (bool div:=false;
    for pd from 2 to k-1 do div:=div ∨ k÷: pd = 0;
    if ¬ div then print (k)fi))
```

Its author has apparently laid the definition of a prime number to heart. A sophisticated compiler might, however, produce code as though the sieve of Eratosthenes had been implemented, thus gaining much efficiency. It is, hopefully, clear that a compiler which would go at such lengths in optimizing the code produced, is likelier to raise the complexity and, thereby, unreliability, than the overall efficiency. This example goes to show convincingly, at least to the author, that there is no such thing as "optimal code", not even as an unattainable ideal that it is worth striving after.

To our mind, the attitude towards code optimization should be the following: First, take care to find solid, general solutions to the problems of run time organization. In doing this, state carefully and clearly the situations for which these solutions have to cater and the conditions which a purported solution must satisfy in order that the problem be solved. In this connection, it is of particular importance, if the solution for problem A depends on some property of that for B, that this property be noted as a condition pertaining to the solution for B also. After all this has been achieved, it is time to think about optimization. The following points have to be kept in mind then:

(i) The class of cases where the optimization should be applicable must be substantial enough to be interesting. E.g., if the code associated with the operator ↑ is very inefficient, it might be interesting to generate for $x\uparrow 2$ the same code as for $x\times x$. On the

other hand, it would be perverse to optimize $x \uparrow 1$ to x, as this would probably occur only in programs for testing this optimization.

(ij) The test for applicability should be simple, not only as an algorithm, but also conceptually. It should be derived as straightforwardly as possible from the conditions for the general solution combined with those for this particular optimization. All too often programming errors are brought about by an unquenchable want for shortcuts; there is little need for automation of this procedure.

(iij) The intended simplification must yield an appreciable gain in efficiency.

(iv) The case for optimization is especially strong when the efficiency of the general solution is weighed down by some feature that the "simple-minded" programmer would not (dare) use or some situation that the honest one would shun. (cf. Bauer's principle, which states that one should not pay for unused (perhaps even unwanted) features. We do not adhere categorically to this criterion: the ALGOL-60-programmer, e.g., who does not use real numbers, should nevertheless accept that procedures treat their "arithmetic" parameters as potentially real.)

In the following, we shall pay attention in particular to those optimizations, whose legitimacy follows from the semantics of ALGOL 68 alone, and which, by virtue thereof, can be applied machine- (and even implementation-) independently.

0.1. Static scope checking.

According to the semantics of ALGOL 68 (8.3.1.2.c Step 1 of the Report), the further elaboration is undefined, when a value is assigned to a name whose scope is larger then that of the value. The reason behind this restriction is the following: The designers of ALGOL 68 have had in mind a practical memory organization, to wit a stack, corresponding to the (dynamical) nesting of the elaboration of ranges. Thus, elaboration of a range may be considered putting a cell, or a number of cells, on top of the stack, all of which will be removed when the elaboration of that range is completed or terminated. In this scheme, some values are meaningful only as long as a certain number of these cells still remain on the stack. Examples of this are presented by the names possessed by local-generators, which may be represented by the address of a cell in the part of the stack corresponding to the range in which they are contained. Other examples are routines, which may give rise to the elaboration of mode-identifiers or operators whose value is to be found in the stack. The reachability of such values outside their domain of meaningfulness has to be prevented effectively (otherwise the further elaboration might be undefined indeed). Therefore, care has been taken that such values cease to exist as soon as one of the parts of the stack on which their meaningfulness depends is removed. This is achieved by ensuring that such values will appear only in parts of the stack which, by the very nature of a stack, must of necessity have been removed then also.

Now the scope of a value is the largest range during whose elaboration the value is meaningful. The restriction that the scope of the name be not larger than that of the value is therefore tantamount to saying that the value may not be represented in a cell other than in the part of the stack which has come to existence during the elaboration of the range which is the domain of meaningfulness of that value. (The implementor is, of course, free to depart from this scheme when he can guarantee otherwise that such meaningless values will become

unreachable.) From these considerations it follows, by the way, that a scope may well be represented by the value of the pointer to the top of the stack at the initiation of the elaboration of the range which is that scope.
So, for reasons of security, each time a value is assigned to a name, a scope check has to be performed. Now there is an interesting class of cases where it can be detected "statically", i.e., at compile-time, that scope checking is superfluous, as can be seen when one realizes that nowhere in the implementation of all of ALGOL 60 the need for scope checking arises.
The idea on which our suggested optimization, viz. the omission of a check on the scopes, hinges, is the association of an "inner" and "outer" scope to some external objects.
Let $\mathcal{D}$ be the destination and S the source of an assignation. If we have determined an outer scope σ for $\mathcal{D}$ and an inner scope τ for S, i.e., scopes σ and τ such that

$$\mathit{scope}\ (\mathcal{D}) \leq \sigma \text{ and } \tau \leq \mathit{scope}\ (S)$$

can be guaranteed both to hold, and it turns out that $\sigma \leq \tau$, then we know that the test "$\mathit{scope}\ (\mathcal{D}) \leq \mathit{scope}\ (S)$" cannot possibly fail and is, therefore, superfluous.

Before we elaborate further on this, we must first obtain some insight into the relationship between such ranges as can be discriminated statically, and the ranges and scopes that may arise dynamically. That the numer of ranges can hardly be bounded statically, is shown by the example

```
(proc r = (int n):if n > 0 then int i:=n; r(i-:=1)fi;
     r(begin int n; read (n); n end))
```

1. Analysis

1.0. Terminology.

In order to facilitate the sequel of this discussion, we shall first develop some terminology.

a) A "prescription model" is a routine-denotation, a procedured-coercend, a procedure-jump or a format-denotation.
(This terminology stems from the fact that prescription models serve as a model for prescriptions, i.e., routines or formats.)

b) An "invocable object" is a prescription model or an actual-declarer which is the actual-declarer of some mode-declaration.

c) The "containing" range of an external object is the smallest range in which that object is contained.

d) An "application" is an identifier (indication, operator) which is an applied (indication-applied, operator-applied) occurrence.

e) The "definition" range of an application is the containing range of the defining (indication-defining, operator-defining) occurrence identified by that application.

f) The scope of a prescription model *P* is the smallest range, if any, in which *P* is contained and which is the definition range of some application contained in *P*, and, otherwise, the program.
(Notice that the difference with 2.2.4.2.b of the Report lies in the fact that here the term "scope" is not defined for a value, but for an external object, the idea being that this scope is the scope of the value the external object will possess upon elaboration.)

g) The "invocation" scope of an invocable object which is a prescription model (an actual-declarer) is its scope (its containing range).

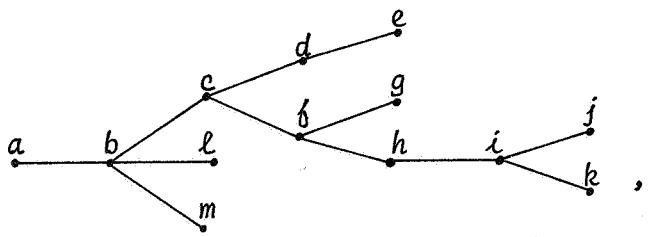
h) An invocable object which is a routine-denotation, procedured-coercend or procedure-jump (a format-denotation, an actual-declarer) is "invoked" by elaborating a closed-clause (the constituent dynamic-replications of a format-denotation, an actual-declarer) derived from it, albeit after some manipulation as described in sections 5.4.2, 6.0.2.d, 7.1.2.b, c, d Step 1, 8.2.2.2, 8.2.3.2, 8.2.7.2.b, 8.4.2 and 8.6.2.2 of the Report.

1.1. The relationship between static and dynamic scopes.

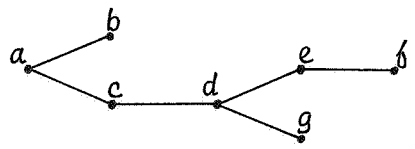
First consider the case without invocable objects. As an example, take

(a:(b:(c:(d:(e:∿)); (f:(g:∿); (h:(i:(j:∿);
(k:∿))))); (l:∿); (m:∿)))

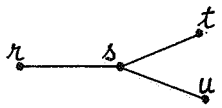
We shall designate each range by a mark bearing resemblance to the representation of its constituent letter-token. We can now draw the following diagram



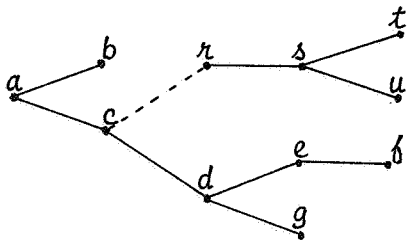
where each range is connected at its right to its constituent ranges, if any. At the initiation of the elaboration of range *h*, say, the ranges which are "active", i.e., being elaborated, are *a*, *b*, *c*, *f* and *h*: the path from *a* to *h* in the diagram. This will be reflected by the state of the stack at that time.
Now suppose that we have a program containing an invocable object, where the diagram of the program, apart from that invocable object, is



The range-nesting structure of the invocable object itself may also be depicted this way, e.g.:

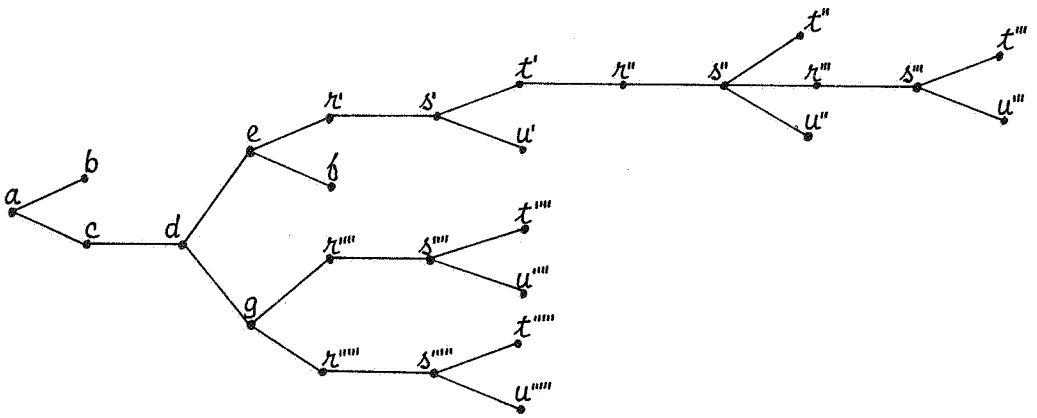


Suppose that the invocation scope of the invocable object is c. We shall indicate this in the diagram as follows:



Notice that the containing range of the invocable object may well be e, say.

Now the invocable object may be invoked in c, and in any range contained in c. A diagram, showing potentially realizable range-nesting through a proper use of invoking, is



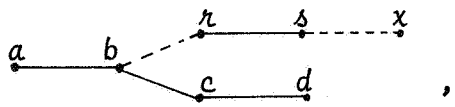
Notice that this diagram presents an example of "recursion": while one invocation (at e) is still active, the same invocable object is invoked again (at t') and once again (at s"). It should be stressed

once more, that at any time only such ranges will be concurrently active as can be found on a path in the diagram from a to some range.

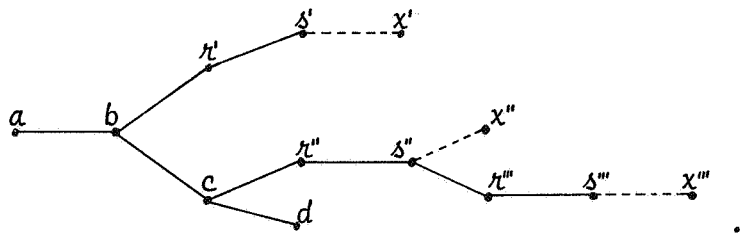
Now suppose that this invocable object contains a prescription model. Then there are two possibilities:

(i) The scope of the prescription model is larger than the invocable object (itself, not its scope!). Notice that in this case the prescription model might as well have occurred outside the invocable object.

(ij) Its scope is not larger. In this case, this prescription model may give rise to different routines or formats, whose scopes depend on the invocation of the embracing invocable object. E.g., the diagram



where r designates an invocable object with invocation scope b, and x a prescription model with scope s, gives potentially rise to the following realizations:



In this example, for each of the invocations at b, c and s'', the value possessed by the prescription model will have a different scope, to wit s', s'' and s''' resp.

So, in general, the scope of the value of an external object will depend on the invocation of the invocable object in which it occurs. However, there is some connection between this scope and the scope

of that object (which has been defined above for prescription models). Consider, as an example, the last two diagrams for the case where we leave the scope of χ, for the moment, unspecified. This scope might be any of the ranges a, b, κ and δ. We can then construct a table of the scope of the value, depending on which invocation it is determined in, and on the scope of the prescription model from which it originated:

scope of the prescription model χ

invocation of κ at

	a	b	κ	δ
b	a	b	κ'	δ'
c	a	b	κ''	δ''
δ''	a	b	κ'''	δ'''

Now the important thing to notice is the following: the "dynamic" scopes have, in each invocation, retained the order of the "static" scopes. It follows that, in order to compare the (dynamic) scopes of the values of two prescription models contained in δ, it suffices to compare the (static) scopes of those prescription models. We claim that this result extends to cases of arbitrary complexity, as can be shown using the following argument:

In order that a prescription model be elaborated, to yield a value, the invocable objects in which it is contained must have been invoked one by one. We shall demonstrate that each invocation leaves the order of the definition ranges of the applications contained in the invocable object unchanged. From this, together with the definition of scope of a prescription model and the definition of the scope of a routine or format given in 2.2.4.2.b of the Report, our claim follows.

Now, consider an invocable object Q with invocation scope σ, two applications contained in Q, and an invocation of Q at some range τ. We have, obviously, $\tau \leq \sigma$. Now we can distinguish three cases:

(i) The definition ranges of both applications are larger than σ. From the semantics of "protecting", it follows that their

definition ranges are not altered by the invocation (*).

(ij) The definition range of one application is larger than σ and the other application has a definition range that is at most equal to σ. Now, the invocation leaves the larger definition range unaltered, whilst the other one becomes τ at most, which in turn is at most σ.

(iij) The definition ranges of both applications are σ or are smaller. Due to the systematic character of possible replacements of identifiers and indications, protection will alter the order of the definition ranges no more than the fact that the modified copy of Q is inserted in the range τ.

(*) This is not wholly true, because of a snag in section 6.0.2.d step 4 of the Report. We shall, however, disregard this.

1.2. Inner and outer scope.

It may be noticed that the ranges of a program constitute a lattice, provided that we introduce a "null scope" ε which is empty, under the two operations

$\sigma \cap \tau$ = the smaller range of σ and τ if one of these is, or is contained in, the other, and, otherwise, ε

and $\sigma \cup \tau$ = the smallest range which is, or contains, both σ and τ (where, by convention, each range other than ε contains ε, whereas ε contains no range).

In this terminology, the inner and outer scope of an external object are a lower and upper bound of the scopes of the future values the external object will come to possess. We shall denote them as a "scope interval" $[\sigma_{inner}, \sigma_{outer}]$.
The scope which is the program will be denoted by "Π" and the containing range of the external object under consideration will be denoted by "ρ".
Furthermore, we define the term "prescope interval" as follows:
if an external object P and an external object Q are one same sequence of symbols, and the original of Q is a direct descendant of that of P, then the prescope interval of P is the scope interval of Q.

We can now give a number of rules to determine a scope interval for an external object P. That the interval given yields indeed safe bounds for the scope of the value of P follows each time from 2.2.4.2 of the Report, the semantics of the Report pertaining to the elaboration of P and the observation of the scope restrictions formulated in 6.1.2.e and 8.3.1.2.c step 1 of the Report, together with the considerations given above. The verification of this is left to the reader as an exercise.
As the rules are listed below in order of their "strength" (where a smaller scope interval is stronger than a larger one), the first rule applicable should be chosen.

(i) If the mode enveloped by the original of *P* is a terminal production of the metanotion 'MODE' after rule 1.2.1.c

"TYPE: PLAIN; format; PROCEDURE; reference to MODE."

has been replaced by the rule

"TYPE: PLAIN."

then the scope interval of *P* is [Π,Π] (or, in words, the scope of any value possessed by *P* is bound to be the program).

(ij) The scope interval of a global-generator (base-vacuum, skip, nihil) is [Π,Π]. (If skips and nihils yields a value to which assignment is impossible, and if at run time a test is performed to detect this case, then the scope interval may be a pseudo-scope-interval; see case xi.)

(iij) The scope interval of a local-generator is [ρ,ρ].

(iv) The scope interval of a prescription model is [σ,σ], where σ stands for its scope.

(v) The scope interval of a mode-identifier (an operator) which identifies the mode-identifier of an identity-declaration (the operator of a caption of an operation-declaration) is that of the actual-parameter of that declaration.

(vi) The scope interval of a dereferenced- (deprocedured-) coercend is [σ,Π], where σ stands for the inner scope of its prescope interval. (If a dereferenced-coercend *D* is an assignation which is a constituent of *D*, then as the scope interval of *D* may be taken the scope interval of its constituent source.)

(vij) The scope interval of a call (formula) is [σ,Π], where σ stands for the greatest lower bound of the inner scopes of its primary (operator) and of those of its constituent actual-parameters (its operands) which are not local-generators.

(viij) When the value of an external object is said to be that of another external object, either explicitly (see, e.g., 8.3.1.2.d step 3 of the Report), or implicitly through 1.1.6.i, then its scope interval is that of the other external object (which, if 1.1.6.i applies, is its prescope interval).

(Notice that rule (v) may be derived by iterated application

of this rule). This rule applies to closed-clauses, united-coercends, assignations, casts and numerous other objects.

(ix) The scope interval of a selection (slice) is that of its secondary (primary).

(x) The scope interval of a rowed-coercend is its prescope interval.

(xi) A jump which is not a procedure-jump has a pseudo-scope-interval, viz. $[\Pi,\varepsilon]$. (The meaning of this pseudo-interval may be grasped by observing that such jumps, when elaborated, will terminate the elaboration of the unitary-clause which they constitute, so that, semantically speaking, no scope violation may occur. Also, the following rule is likely to shed some light on its significance.)

(xij) The scope interval of a serial- (collateral-, conditional-) clause is $[\sigma,\tau]$, where $\sigma(\tau)$ stands for the greatest lower bound of the inner scopes (the least upper bound of the outer scopes) of the units of its constituent clause-trains (its constituent units, the then-clause and the else-clause of its choice-clause). (This rule may be considered "balancing" of scopes.)

(xiij) The scope interval of an external object (when all other rules fail) is $[\rho,\Pi]$.

The static scope check for a serial-clause now reads: Determine its inner scope. If that inner scope is larger than that serial-clause, then the dynamic scope check (6.1.2.e of the Report) may safely be omitted.

The static scope check for an assignation now reads: Determine the outer scope of its destination, and the inner scope of its source. If that outer scope is not larger than that inner scope, then the dynamic scope check(8.3.1.2.c Step 1 of the Report) may safely be omitted.

2. Concluding remarks.

From a number of relatively simple rules static scope checks can be derived, which in the large majority of cases in ordinary run-of-the-mill programs will make dynamic scope checking superfluous. It should be noted, however, that the security offered by the scope restrictions will only then be fully effective, when it cannot be invalidated by the result of elaborating a mode-identifier or a formula having an operator whose corresponding declaration has not yet been elaborated. Therefore, as a part of the implementation, the initiation of a serial-clause should entail making all mode-identifiers and operators of which it is the definition range, possess a value whose scope is in accordance with the scope interval determined for those mode-identifiers and operators.