Partial Evaluation and Mixed Computation
D.Bjørner, A.P.Ershov and N.D.Jones (Editors)
Elsevier Science Publishers B.V. (North-Hollabd)

THE ALGORITHM OF GENERALIZATION IN THE SUPERCOMPILER*

Valentin F. Turchin

Computer Science Department
The City College of New York
New York, N.Y 10031 USA

The central problem of supercompilation is to find a finite set of configurations (generalized states) of the computing system which is, for a given initial configuration, self-sufficient in the sense that the process of computation can be defined by a finite graph of states and transitions using only these configurations as nodes. Generalization over configurations is necessary for this. The paper describes an algorithm of generalization in the process of outside-in driving (forced unfolding of function calls in the lazy evaluation semantics) which always terminates and produces a finite graph of states and transitions with a self-sufficient set of basic configurations.

1. INTRODUCTION. WHY TO GENERALIZE?

It may seem strange that the problem of generalization is raised in the context of partial evaluation. Indeed, partial evaluation is mostly used for, and therefore perceived as, program specialization, and this is something opposite to generalization.

However, we discuss here a special technique of function transformation, which is referred to as *supercompilation* (see [1-3]). When supercompilation is used for the sake of partial evaluation (which is not always the case, because supercompilation can do more) it comes to the specialized program in a different way than the straighforward partial evaluation.

In partial evaluation we have an original, general, program, and a specialized function call. Then we make a global analysis of known and unknown arguments, and specialize the original definition step by step, watching that a certain limit is not overstepped. Thus the loops in the specialized program are the old loops of the original program, but (possibly) specialized. Partial evaluation technique is, in a sense, monotonous with respect to specialization.

In supercompilation we, again, have an original, general, program, and a specialized function call. Here, however, we never specialize the original program. We start from the ultimate specialization of the initial call, and then construct a program for it by *driving*. If the program can be made self-contained without looping back (a simple tree), there will be no generalization necessary. Usually, however, we have to loop back, and these are *new* loops, created *ad hoc* for current configurations. This may make it necessary to generalize configurations, because the former configura-

* This work was supported by the National Science Foundation under grant DCR-8412986.

tion will not always be general enough. Thus the process of supercompilation is not monotonous: we first jump to the completely specialized initial call, considered as the initial (degenerate) graph of states and transitions, and then develop it into a self-contained graph i.e. a program, using generalization when necessary.

While partial evaluation has the narrow goal of specializing functions, supercompilation is a much wider framework for general function transformations. We believe that it follows closer than other techniques to the way we, human beings, think. Thinking is creating mental models of the processes in the world around us. How do we create those models? We watch the processes and try to form some generalized states of the explored systems in terms of which we can construct a self-sufficient model of the processes, i.e. represent the processes as transitions between the basic generalized states. But this is exactly what the supercompiler is doing.

2. HOW TO GENERALIZE?

As an introduction to the problem of generalization, consider this example. Suppose, two strings are given:

```
'ABA'
'ABXYABA'
```

and we are asked to write a generalization which is, in some intuitive sense, the best. Then we should ask, before anything else, what is meant by a generalization? The first step to define a generalization is to notice that a generalization of a number of objects is a set which includes all of these objects. This definition is not sufficient, however, because then the best generalization in our example would be simply the set of exactly the two strings mentioned, and a similar trivial solution would exist in any situation. Actually, when we speak of generalizations, we have in mind a language in which sets of objects are defined, and we want not just a set of objects, but an expression of this language defining a set of objects which includes all the objects to be generalized -- and, possibly, some other objects. Then the problem of a "good" generalization is non-trivial.

Let the language to describe sets of strings be that of simple patterns, as in Refal, where s1, s2 etc. stand for single symbols, i.e., in our context, letters of the alphabet, and e1, e2, etc. stand for arbitrary expressions -- here for strings, including the empty string. Thus, 'A'e1 is the set of strings starting with 'A'; e1 s2 s2 is a string ending with two identical letters, etc. Then for the two strings above, even after we exclude those generalizations for which we see obviously better (tighter) generalizations, we still have quite a number of reasonable solutions, for example:

```
(1)     'AB'e1
(2)     'AB's1 e2
(3)     e1'ABA'
(4)     'AB'e1'A'
```

Which one to choose?

We faced this problem when working on the Refal supercompiler, because intelligent generalization is the central problem of su-

percompilation. We do not discuss here the concept of a supercompiler in detail; the reader can address [1], or [2], or an earlier and detailed (but not so easily available) publication [3]. The objects to be generalized in supercompilation are function calls in Refal. Experimentation with different ways of generalization led us to the following principle, which we believe to be of universal significance for symbolic objects:

The Generalization Principle. Generalization of objects has a meaning only in the context of some processes of computation in which the objects take part. Then the language of generalization should have means to describe computation histories, and generalizations should be sets of objects which have common computational histories up to a point.

According to this principle, we should not generalize unless we know in what computational processes our two strings are taking part. If we know, for instance, that the strings are scanned form left to right, then the appropriate series of generalizations, each next being tighter than (a subset of) the preceding, will be:

```
s1 e2
'A'e2
'A's1 e2
'AB'e2
'AB's1 e2
```

Thus if we want the tightest generalization, we take the last one. Should the strings be processed differently, the generalizations would be defined differently. If no algorithmic processes are defined over strings, there is no sense in generalization.

In the following sections of this paper we describe the algorithm of generalization in the supercomiler based on this principle. In the context of the language we use in the supercompiler, namely, Refal, computation histories become tangible formal objects. It should be noted that Refal fits the needs of generalization on two counts. First, it has the concept of a pattern, which is, of course, the simplest form of generalization, built into the language. Second, the functioning of the Refal machine is a simple sequence of substitutions, which facilitates the formalization of computational histories.

3. NEIGHBORHOODS

The objects we deal with in supercompilation are Refal graphs, which are, essentially, graphs of states and transitions of the Refal machine. The nodes of a Refal graph are Refal expressions, the edges (directed) are transformations of two kinds: contractions and assignments. Both are pattern-matching operations over variables, with the variables in the left-hand side having some values, and the variables in the right-hand side being defined by the operation. A contraction has a single variable in the left side, and a pattern in the right side, e.g.

```
e1  →  s2 e1
```

is a conditional operation which checks that the value of e1 starts with a symbol on the left, assigns that symbol to s2, and redefines e1 as the remaining part of the original value. An assignment has a single variable on the right and defines its new

value through constants and the variables of the left side, e.g.

```
'A'eX s2(eY)  ←  eX
```

A Refal program can be represented as a Refal graph defining one step of the Refal machine, e.g. the program:

```
FAB {el = <FAB1 ()el>; }
FAB1 {
      (el)'A'e2 = <FAB1 (el'B')e2>;
      (el)s3 e2 = <FAB1 (el s3)e2>;
      (el) = el;
     }
```

is, essentially, the graph:

```
:( eØ → <FAB el>; <FAB1 ()el> ← eØ
 + eØ → <FAB1 el> :( el → (el)'A'e2; <FAB1 (el'B')e2> ← eØ
                   + el → (el)s3 e2; <FAB1 (el s3)e2> ← eØ
                   + el → (el); el ← eØ
                   )
 )
```

Here we used the form $:(B_1+\ldots+B_n)$ to represent n branches $B_1 \ldots$ etc., which start from the same node. The nodes themselves are left out in this graph; they can be restored when reading the graph. Refal graphs are read as follows. The variable eØ stands always for the content of the view-field (the current expression being transformed) of the Refal machine. Our graph consists of two subgraphs. The first begins with the contraction eØ → <FAB el>, which corresponds to the case where the expression in the view-field of the Refal machine is a call of the function FAB with a completely unspecified argument represented by the free variable el. The state of the view-field at this moment is, obviously, <FAB el>; we skip it. The next operation is the assignment to the view-field eØ of a new value, which is a call of FAB1; we can skip the node again, without losing information. When we construct the graph of states for an arbitrary expression in the view-field eØ, we need not write out nodes explicitely, because the current node is always identical to the current value of eØ.

The second subgraph is a definition of the function FAB1. Here we separated the general configuration of the call of a given function, <FAB1 el>, from the detalization provided by sentences. This gives us our first insight into the concept of a *neighborhood*. The first thing the Refal machine does to perform a step is to identify a function symbol, which should follow the left evaluation bracket <. Thus <FAB 'ABC'> and <FAB 'XY'> appear the same for the Refal machine at this stage; they belong to the same neighborhood <FAB el>. Any call of FAB1 belongs to a different neighborhood, namely <FAB1 el>. Inside this neighborhood we see a further differentiation: <FAB1 ('X')'ABC'> and <FAB1 ('PQ')'AC'> are indistinguishable to the Refal machine as long as it executes one step on them: in both cases the first sentence is used. The expression <FAB1 ('XY')'BCD'>, however, will be distinguished in the first step from those two. The former neighborhood is <FAB1 (el)'A'e2>, the latter <FAB1 (el)s3 e2>, with the restriction that s3 is not equal to 'A'.

Complex contractions which we find in the left sides of Refal sentences can be decomposed into simpler contractions. In the example above, the left side of the first sentence of FAB1 was

decomposed as follows:

```
e0 → <FAB1 (e1)'A'e2>  =  e0 → <FAB1 e1>; e1 → (e1)'A'e2
```

We could go further and decompose it into

```
e0 → <FAB1 e1>; e1 → (e1)e2; e2 → 'A'e2
```

Contractions are elements of computation histories. The more we decompose contraction, the more detailed the description of histories will be. This process comes to its natural close if we decompose all left sides of Refal sentences into *elementary* contractions. There are seven of these, namely:

1. `eX → sY' eX`
2. `eX → (eY')eX`
3. `eX → eX sY'`
4. `eX → eX(eY')`
5. `eX →`
6. `sX → S`
7. `sX → sY`

Here S stands for a definite (but arbitrary) symbol, and the primed variables sY' and eY' symbolize that the index Y' of the variable is *new*, i.e. was not used before.

The decomposition of the left side above into elementary contractions is:

```
e0 → <FAB1 (e2)'A'e1>  =

e0 → <FAB1 e1>; e1 → (e2)e1; e1 → s3 e1; s3 → 'A'
```

(We renamed some variables in the left side; this, of course, changes nothing).

Definitions. An expression without free variables is a *ground* expression. We say that a contraction is executed *positively* over a ground expression, if the contraction is found applicable and applied; we say that it is executed *negatively* if it is established that the contraction is not applicable. The sequence of elementary contractions executed positively or negatively over a ground expression in *n* steps of the Refal machine is its *computation history* of *n*-th order. The set of all ground expressions with

a common computation history of *n*-th order is a *neighborhoood* of *n*-th order.

Thus to every computation history a neighborhood corresponds. We shall denote neighborhoods by the same symbols as histories. If a history H_1 is a prefix of H_2, then the neighborhood H_2 is a subset of H_1. This relation between neighborhoods is a partial order.

A Refal program defines a system of partially ordered neighborhoods, in other words, a topology, in the space of ground expressions. The longer is the common part of computation histories

of two points in this space, the tighter is their common generalization to a neighbothood, in other words, the closer are these points. Note that speaking of ground expressions we have in mind only *active* ground expressions, i.e. those including at least one pair of activation brackets. All *passive* expressions fall in one big class with a zero-length computation history, and are of no concern to us. This is, of course, a consequence of the generalization principle formulated above.

A compact representation of a neighborhood as a set can be obtained by folding the contractions of the corresponding history into one pattern. With the program above, the system of first-order neighborhoods is as follows:

```
(a)       <FAB e1>
(b)       <FAB1 e1>
(c)       <FAB1 (e2)e1>
(d)       <FAB1 (e2)s3 e1>
(e)       <FAB1 (e2)'A'e1>
(f)       <FAB1 (e2)s3 e1> (#s3 → 'A')
(g)       <FAB1 (e2)>
```

The *restriction* (negative contraction) in (f) indicates that only those ground expressions are in the pattern in which s3 is distinct from 'A'. These neighborhoods are partially ordered as follows:

```
b > c > d > e
        d > f
    c > g
```

where > denotes being a superset.

To compute the neighborhoods of the second order, we use *driving* (see, e.g., [1]). Driving every active end-node in the graph for FAB1, we come to the graph that represents two steps of the operation of the Refal machine if it starts with any call of FAB1. It contains all possible computation histories of length two. Six new neighborhoods will be added to the system. Three of them are refinements of (e):

```
(h)       <FAB1 (e2)'AA'e1>
(i)       <FAB1 (e2)'A's3 e1>      (# s3 → 'A')
(j)       <FAB1 (e2)'A'>
```

and the other three, analogously, develop (f).

Driving can be repeated as long as there are active end-nodes in the graph. We refer to this process as *exhaustive driving*. It can, and typically will, go on infinitely. Exhaustive driving defines the set of *ultimate neighborhoods*, which correspond to terminated computation histories. In the case of FAB1 the ultimate neighborhoods are:

```
(1)  <FAB1 (e2) >
(2)  <FAB1 (e2)'A'>
(3)  <FAB1 (e2)s3>              (# s3 → 'A')
(4)  <FAB1 (e2)'AA'>
(5)  <FAB1 (e2)'A's3>           (# s3 → 'A')
(6)  <FAB1 (e2)s3'A'>           (# s3 → 'A')
(7)  <FAB1 (e2)s3 s4>           (# s3 → 'A') (# s4 → 'A')
... etc.
```

The expressions which belong to the same ultimate neighborhood pass through the Refal machine in the exactly identical ways; the machine has never a chance to discover the difference between them.

4. WHEN TO GENERALIZE?

The idea of a supercompiler is to superwise the construction of the full graph of states for the initial configuration, and at certain moments loop back, i.e. reduce an end-configuration -- directly, or with a generalization -- to one of the previous configurations, and in this way construct a finite graph on the basis of a potentially infinite process. A direct reduction is possible when the later configuration is a subset of the earlier one. This is an easy case, when it is pretty obvious that the reduction can be made and has sense. The difficult case is when the later configuration is not a subset of the previous one, but is "close" to it in some sense. If we simply ignore this closeness, and go on with driving, we may never loop back, and the process will never stop.

Take a simple example with the functions we defined above. We want to supercompile the configuration

(1) <FAB e1>

Nothing especially interesting is expected here. The supercompiler must simply return the original definition. Our purpose is to see that the supercompiler can indeed find the correct basic configurations for looping back whenever necessary to terminate the work.

The graph of states we construct in supercompilation must include nodes, i.e. configurations of the Refal machine, explicitely, because we want to compare and generalize configurations. Let the nodes in graphs be represented by references to configuration definitions. The first step of driving replaces (1) by the call of FAB1, so the graph is the unconditional transition:

(1) (2)

with the definition:

(2) <FAB1 ()e1>

Next step of driving results in the graph:

```
(1) (2) :( e1 → 'A'e1; (3)
         + e1 → s2 e1; (4)
         + e1 → []; (5)

(3) <FAB1 ('B')e1>
(4) <FAB1 (s2)e1>
(5) []
```

(For readability, we use [] to represent the empty expression).

The passive configuration (5) terminates the walk in the graph. None of the new active configurations (3) and (4) is a subset of any of the previous configurations (1) and (2). If this were our criterion for looping back, we would go on with driving. After the

next steps we would have such configurations as

(6) <FAB1 ('BB')e1>
(7) <FAB1 ('B's2)e1>

etc., none of which, again, would loop back onto any of the previous configurations. In this way we would never come to a finite graph.

To loop back properly, we must recognize that (3) and (2) are close enough for looping back. Indeed, they belong to the same first-order neighborhood

(N) <FAB1 (e2)e1>

If we set as a principle that belonging to the same first-order neighborhood is a sufficient reason for looping back, we generalize (3) and (2) to (N), express (2) through (N):

```
(2) = [] ← e2; (N)
```

and recompute the graph for the generalized configuration (N):

```
(1) [] ← e2; (N) :( e1 → 'A'e1; (3')
                  + e1 → s3 e1; (4')
                  + e1 → []; (5)
```

(3') <FAB1 (e2'B')e1>
(4') <FAB1 (e2 s3)e1>

Now (3') and (4') are subsets of (N); reducing them to (N) we come to the graph

```
(1) [] ← e2; (N) :( e1 → 'A'e1; 'B'e2 ← e2; (3')
                  + e1 → s3 e1; e2 s3 ← e2; (4')
                  + e1 → []; (5)
```

Our algorithm of generalization is based on keeping in memory the first-order neighborhoods of past configurations. We formulate it first for the case where all function calls have passive arguments only, i.e. there are no nested calls. Nested calls will be considered in the next section.

As the Refal machine applies to the function argument one elementary contraction after another, the neighborhood that describes the function call becomes more narrow. Then the replacement is executed, another descending sequence starts, etc. We have the following row of neighborhoods in each branch of the graph:

$$f_1^1 \quad f_2^1 \ \ldots \ R^1 \quad f_1^2 \quad f_2^2 \ \ldots \ R^2 \quad \ldots \quad f_1^n \quad f_2^n \ \ldots \quad f_m^n$$

They are partially ordered as follows:

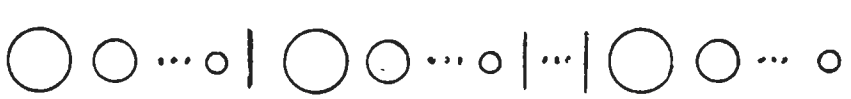
$$f_1^1 > f_2^1 > \ldots$$

$$f_1^2 > f_2^2 > \ldots$$

$$\ldots$$

$$f_1^n > f_2^n > \ldots \quad f_m^n$$

In a graphic form:



There are several variants of the algorithm, which place the resulting program in different positions on the compilation-interpretation axis (the more detailed is the set of basic configurations, the more compilative the program; the more general the basic configurations are, the more interpretive the program, see [1]). The most interpretive variant is as follows. Each time before we make the next replacement, R^n, we compare each neighborhood of the current step, starting with the first one, E^n_1, with all the previous neighborhoods, moving from R^{n-1} backwards, to the beginning of the walk. If we find the same neighborhood, we loop back to it. In this way we find the most general from the recurring neighborhoods. If we loop back, R^n is ignored and the step due is not executed; reduction takes place instead. Since the number of different first-order neighborhoods is finite, the algorithmic process is always finite.

This algorithm can be obviously generalized for neighborhoods of an arbitrary order. The higher the order, the more compilative will the resulting program be. The same effect can be achieved by function iteration, using only the first-order neighborhood algorithm. If we define functions that correspond to two, three, etc. steps of the Refal machine, and use the first-order algorithm with them, then this will be equivalent to higher-order neighborhoods for the original system of functions. We can control the process of generalization by iterating some functions, while leaving alone others. Therefore, the algorithm based on first-order neighborhoods has a certain property of completeness. If we accept the principle that the closeness of expressions should be measured by the length of the common part of their computation histories (the program-induced topology), then all strategies of generalization can be presented as refinements of an algorithm based on first-order neighborhoods.

5. GENERALIZATION OF NESTED CALLS

If nested function calls are executed according to the inside-out principle, known also as the applicative evaluation order, then the computation of every active expression can be broken down into a sequence of computations and substitutions, this sequence being independent of function definitions. For example, the assignment

```
<F e1 <G e2> <H e3>>  ←  eØ
```

will be decomposed into the sequence of assignments:

```
<G e2> ← eX; <H e3> ← eY; <F e1 eX eY> ← eØ
```

We shall refer to such decompositions as *stacks*. Since the order of execution is strictly left-to-right, computation histories -- and, therefore, neighborhoods -- for stacks break into pieces corresponding to the first, second, etc. segments of the stack. If a stack S_1 is a prefix of another stack, S_2, then the neighborhoods of S_1 are supersets (generalizations) of the neighbor-

hoods of S_2. There is no interaction between neighborhoods corresponding to different segments of the stack.

In the supercompiler, however, we use the outside-in (normal, lazy) order of evaluation, because it provides one of the primary means of optimization. In this case the situation is much more complicated. A prefix of a decomposition is still a generalization of a longer decomposition, of course. But we cannot decompose a nested call into a stack without consulting function definitions. The decomposition is still made, but it is made in the process of moving from outside in, and it may depend on the values of variables. Computation histories may consist of alternating pieces from different function calls. Indeed, suppose that the computation process starts with the all-embracing function call, but after executing a number of contractions the Refal machine finds that a not yet computed call inside is a hindrance for further application of sentences. Then it will leave the unfinished function call as a *context*, and switch to the computation of that internal call, which, in turn, may send the machine further inside. After computing the internal call -- completely or partially -- the process returns to the point in the outer function call where it was interrupted.

Let us describe this in somewhat more detail. We call an expression *unitary active*, or just *unitary*, if it is of the form <*E*>, where *E* is any expression (possibly active, so that there are nested function calls). If the result of replacement in the execution of a Refal step is unitary, we make it our next active subexpression to compute. If it is not unitary, it is either passive (completed computation), or non-unitary active (partially computed, with some passive parts outside of activation brackets, e.g. 'A'<FAB e1>). In both cases we substitute the result into the context, and take the context as the next active subexpression to compute. If there is no context (bottom of the stack call) and the result of the step is passive, this is the end of driving. If the result is partly passive, the passive part is kept in the viewfield of the Refal machine, and the unitary active part is driven further.

We shall consider a few examples which typify different structures of recursion. We shall demonstrate how we come to our algorithm of generalization, and how it works. Then we shall prove that this algorithm has a guaranteed termination.

The first example is the classical recursive definition of the factorial:

```
FACT {0 = 1;
      1 = 1;
      sN = <MULT sN <FACT <SUB sN 1>>>;
     }
```

We assume that the arithmetic functions SUB and MULT are built-in (not defined in Refal) functions which require their arguments to be ready-for-use numbers. Then the inside-out and outside-in orders of evaluation will lead to the same sequnese of operations. We see here three neighborhoods involved:

```
(f)  <FACT s1>
(m)  <MULT s1 s2>
(s)  <SUB s1 s2>
```

(To simplify things, we ignore such neighborhoods as <FACT e1>, <FACT s1 e2>, etc., which cause unique transitions). A stack will be denoted as a string of neighborhoods, e.g., sfm will stand for any of the nested calls like that in the definition of FACT.

When we simply drive <FACT s1> exhaustively we have, on one of the branches, the sequence of neighborhoods:

f; sfm; fm; sfmm; fmm; sfmmm; fmmm; ... etc.

which goes on infinitely. Let us now apply the simple algorithm of comparing neighborhoods which we developed for the case of one-level function calls. We extend it by recalling that a stack is a specialization (subset of) its every prefix. At the third stage of the process above we recognize that fm is a subset of f. Thus we declare f basic, and come to the original algorithm.

This experience suggests to accept as the general criterion of generalization a situation where the current stack is of the form XY, where X is a previous stack. This criterion, of course, includes the one-level situation as a special case where Y is empty and X is one segment.

However, if we only slightly change our example, this criterion will not work. Let the factorial function be computed in the context of some other function, say,

(*) <ADD 1 <FACT sN>>

If we denote by a the neighborhood corresponding to ADD, the sequence of stacks in driving will be:

fa; sfma; fma; sfmma; fmma; sfmmma; fmmma; ... etc.

One can see that none of the previous stacks is a prefix of a subsequent one. Therefore, the process will never terminate.

The reason for this failure is that the algorithm, as it is at this point, does not draw a line between the part of stack that is recurrent, and the part that does not really participate in action, but is a passive context. We, therefore, modify the algorithm as follows. The stack will not be just a linear segment, but a structure of parenthesized segments, where the context part is taken outside of parentheses. Accordingly, the computation history will be written in such a way that the context is left outside of the parentheses as a common part to all the stages of the process as long as it has no impact on developments.

The nested call (*) will now be characterized by the formula (f)a. It results from outside-in driving, where we start driving from the call of ADD, an then see that before anything is done on this call, we must drive FACT. So, we leave ADD as a context, and FACT becomes the active subexpression.

After the first step of the Refal machine, the history of computation takes the form:

(f; ((s)f)m)a

Then SUB is computed, and the next history record will be:

```
(f; (sf; f)m )a
```

We have followed here the Orwellian principle of permanently rewriting the history. We have a better reason, though, than in Orwell's novel. When s is computed, the result is substituted into f; thus the real previous state to be used in comparisons should now be seen as sf, not (s)f. Each time that a context enters the play, we open the parentheses that separate it from the active part at the current stage and all previous stages of history since this context appeared.

As before, we compare the last stack with all the previous stacks at every stage of development. When we exit context parentheses while tracing the history backwards, we add the context to the current stack before comparing it with next previous stacks. So, after the first step of the Refal machine, we compare sfm with f. After the second step we compare f with sf, and then fm with f. The last comparison discovers that f is a repeated prefix, and the algorithm successfully terminates.

Consider one more example. Let F be the function that scans the argument from left to right and replaces each pair of identical symbols by one symbol of the same kind:

```
F {
   s2 s2 e1  =  s2 <F e1>;
   s2 e1  =  s2 <F e1>;
     = ;
  }
```

Let the initial configuration be

```
1.        <F <F <F e1>>>
```

We want to supercompile it using, as always, the outside-in order of evaluation, so that the final program performs in one pass the job which is defined by the initial configuration as a three-pass job. In this problem, it is easy to discover that the same function F is called again and again by itself, and declare it basic. But if we do so, we, obviously, return to the original three-pass program. The problem here is of just the opposite kind: how to delay looping back in such a manner that the result is a one-pass program. The algorithm must steer carefully between the Scylla of looping back too early, and the Charybdis of never looping back at all. We are going to show that our algorithm is capable of this navigational feat.

Let us concentrate on the first branch in every step of driving. Should we drive manually, we would produce this sequence of nodes:

```
2.        <F <F s2<F e1>>>
3.        <F <F s2 s2<F e1>>>
4.        <F s2<F <F e1>>>
5.        <F s2<F s2<F e1>>>
6.        <F s2<F s2 s2<F e1>>>
7.        <F s2 s2<F <F<F e1>>>
8.        s2 <F <F <F e1>>>
```

At this stage, we would notice that the initial configuration reappears at the top level. We would separate it and terminate the branch. We want now to see how the supercompiler will do this.

There are three neighborhoods at work in this example, which will be denoted as a, b, and c:

(a) <F eX>
(b) <F s2 eX>
(c) <F s2 s2 e1>

Let us trace how the history changes while the supercompiler works. The initial history is

1. ((a)a)a

There is no semicolon here, which signifies the fact that no step has yet been made. We simply decomposed the initial configuration into a stack. We shall now go through the stages 1 - 8 of driving above, using the stack-of-neighborhoods notation.

In the first step of the Refal machine, we use the contraction:

e1 → s2 s2 e1

The replacement results in s2<F e1>. We now have the node

<F <F s2 <F e1>>>

Driving it outside-in, in order to decompose it into a stack, we find both the first, and the second call of F impossible to complete, so the active subexpression will be the third F again. The decomposition is:

<F e1> ← eX; <F s2 eX> ← eY; <F eY> ← eØ

In the short notatation,

((a)b)a

Since the second F from outside (the context of the active third F) takes part in this transformation, we must open the corresponding parentheses: it is not just a which becomes b, but aa which becomes (a)b. Thus on the second stage the computation history is:

2. (aa; (a)b)a

When we compare the current situation with every stage of history, we do not exit from the subgraph common to both. So, what we actually compare at this stage is ab with aa. The result is negative, and we go on. After the second step the node is

<F <F s2 s2<F e1>>>

Driving from outside in, we find the second F to be the active subexpression. The third F is not seen by the Refal machine; the neighborhood formula is (c)a. Since the context, which is now b, has taken part in the process again, we open the parentheses, and the history becomes:

3. (aa; ab; c)a

Procedeing in this manner, we produce the further members of the "history of histories":

```
4.   aaa; aba; ca; ((a)a)b
5.   aaa; aba; ca; (aa; (a)b)b
6.   aaa; aba; ca; (aa; ab; c)b
7.   aaa; aba; ca; aab; abb; cb; c
8.   aaa; aba; ca; aab; abb; cb; c; ((a)a)a
```

Nowhere in the history before the last stage did we see a repeating context, so the process went on. At the last stage ((a)a)a compares positively with aaa, and this combination is declared basic. One can see that on all branches of the graph a similar situations take place, so that in the end we have a finite graph.

Our last example is the merge-sort algorithm, which illustrates one more pattern of recursion.

```
SORT { el = <CHECK <MERGE <PAIRS el>>>; };

MERGE {
  (el)(e2)eR = (<MERGE2 (el)(e2)>) <MERGE eR>;
  (el) = (el);
    =  ;
    };

CHECK {
  (el) = el;
  el = <CHECK <MERGE el>>;
     };
```

We shall not use the definitions of the functions PAIRS and MERGE2. The former makes up the initial list of pairs from the input list of items, which are assumed to be, syntactically, Refal symbols (e.g., numbers). The latter merges two lists. We assume that PAIRS has been executed, so that the initial configuration is

```
1.   <CHECK <MERGE el>>
```

where el is a list of pairs.

Driving this configuration outside-in, we have the following row of configurations in the branch where el in the argument of MERGE is not yet exhausted. We write C and M for CHECK and MERGE, and put the ellipsis instead of MERGE2 calls, which make no impact on driving:

```
2.   <C (...)<M el>>
3.   <C (...)(...) <M el>>
4.   <C <M (...)(...) <M el>>>
5.   <C (...)<M <M el>>>
6.   <C (...) <M (...) <M el>>>
7.   <C (...) <M (...)(...) <M el>>>
8.   <C (...)(...) <M <M el>>>
9.   <C <M (...)(...) <M <M el>>>>
10.  <C (...) <M <M <M el>>>>
```

The neighborhoods involved are:

```
(m)      <MERGE el>
(m1)     <MERGE (e2) el>
(m2)     <MERGE (e3)(e2) el>
(c)      <CHECK el>
(c1)     <CHECK (e2) el>
(c2)     <CHECK (e3)(e2) el>
```

The problem with this type of recursion is that the function CHECK is not a passive context, but one of the functions responsible for recursion; it cannot be taken outside of parentheses. If we look at the states of the stack at the moments when e1 is tested, i.e. 2, 5, 10, etc., we see the sequence:

$$mc_1;\ mmc_1;\ mmmc_1;\ \ldots \text{ etc.}$$

where no stage is a prefix of any subsequent stage.

Nevertheless, our algorithm discovers the potential infiniteless of recursion, and declares <CHECK (e3)(e2)e1> a basic configuration. We leave it to the reader to verify that the computation history will develop as follows:

1. (m)c
2. $mc;\ (m)c_1$
3. $mc;\ mc_1;\ c_2$
4. $mc;\ mc_1;\ c_2;\ (m_2)c$
5. $mc;\ mc_1;\ c_2;\ m_2c;\ ((m)m)c_1$
6. $mc;\ mc_1;\ c_2;\ m_2c;\ (mm;\ (m)m_1)c_1$
7. $mc;\ mc_1;\ c_2;\ m_2c;\ (mm;\ mm_1;\ m_2)c_1$
8. $mc;\ mc_1;\ c_2;\ m_2c;\ mmc_1;\ mm_1c_1;\ m_2c_1;\ c_2$

At this stage the stack c_2 repeats itself, and the supercompiler declares it basic.

6. TERMINATION OF THE ALGORITHM

We want to prove now that the algorithm we have outlined and illustrated above always leads to a finite graph, because the driving of every branch of the graph will terminate, either because the resulting node is passive, or because the current stack has one of the previous stacks as its prefix (looping back). To formulate our algorithm in exact terms and to prove its termination, we must first review the formal objects which are used in the algorithm.

We represent the nodes of the graph of states by *stacks*, which consist of *neighborhoods* and are used in two forms: with and without parentheses. The current stack, as it appears from a step of the Refal machine, is represented in fully parenthesized form, which can be described by the following BNF:

```
C-ST  ::=  empty  |  '(' C-ST ')' f
```

Here quoted objects stand for themselves, and unquoted objects are classes of objects. The bar | separates alternatives. C-ST is a current stack, and f a neighborhood (function call). In our examples above, the neighborhoods were represented by letters.

When stacks stand for the past states, however, they are represented by strings of neighborhoods, which reflects the fact that these neighborhoods took part in the computation and must be considered together as representing one composite configuration of the Refal machine. Thus we introduce *past stacks*, which make up the class of objects STACK:

```
STACK  ::=  empty  |  STACK f
```

The consecutive members of computation histories are separated by semicolons, hence we need *history segments*, class H:

```
H  ::=  empty  |  H STACK ';'
```

As a result of maintaining the history records at every parenthesis level of the current stack, the overall record, which we shall designate as *the ongoing history*, ON-HIS, is from the class:

```
ON-HIS  ::=  H  |  H '(' ON-HIS ')' f
```

In a more reviewable form, the ongoing history is:

(*) $H_0(\ldots(H_{n-1}(H_n\ f_n)f_{n-1})\ldots)f_0$

where each H_i is a history segment, and f_i a neighborhood.

Now every branch of the graph of states which is being constructed by driving has a formal representation as an ON-HIS. The next thing to do is to formulate the rules according to which the ongoing history is transformed in driving, and define in exact terms the conditions under which a given branch is cut off, either because of the termination of driving, or because of looping back to a past stage. After that we shall be able to prove that under those condition no ON-HIS, i.e. no branch in the graph, can be infinite.

The starting point of driving is a current state C-ST which represents the initial configuration of the Refal machine. There are three transofrmation rules for ON-HIS. To put them as replacement formulas, we denote objects by the same symbols as the BNF classes to which they belong, adding subscripts when necessary.

Transformation Rules for ON-HIS

T1. Active replacement rule

$$H_n\ f_n \quad \text{-->} \quad H_n\ f_n;\ \text{C-ST}$$

T2. Passive replacement rule:

$$H_{n-1}\ (H_n\ f_n)\ f_{n-1} \quad \text{-->} \quad H_{n-1}\ H_n{*}f_{n-1}\ f_n f_{n-1};\ \text{C-ST}$$

T3. Termination rule:

$$H_n\ f_n \quad \text{-->} \quad H_n$$

Here f_n stands for the current active (top of stack) neighborhood, and H_n is the immediately preceding history segment. The active neighborhood in an ON-HIS is located as the one just before the first right parenthesis. The operation H*f in Rule T2 is the distribution of a neighborhood over a history segment defined by the formula:

$$H{*}f = [\text{STACK}_1;\ \text{STACK}_2;\ \ldots\ \text{STACK}_k]{*}f =$$

$$\text{STACK}_1 f;\ \text{STACK}_2 f;\ \ldots\ \text{STACK}_k f;$$

When a step of the Refal machine is performed in the process of driving, one rule must be applied to the ON-HIS representing the current branch. The three transformation rules correspond to the

three cases in the Algorithm of outside-in driving above. If the result of the step is a unitary active expression, Rule T1 is applied. According to this rule, the current active neighborhood is added to the history of computation on its parenthesis level, the context remains unchanged; a new Current Stack C-ST results form the step. If the result of the step is passive or non-unitary active, and there is a context (i.e. $n > \emptyset$), Rule T2 is applied. In this case one level of parentheses is eliminated; the context neighborhood f_{n-1} is added to each stack in the history segment H_n; one more History Stack, $f_n f_{n-1}$, is added to the history, and followed by a semicolon; then new C-ST appears. If there is no context and the result of the step is passive, Rule T3 is used. It terminates the branch. In case of a non-unitary result and $n = \emptyset$ (no context) Rule T1 is used.

The Cut-Off Rules

C1. Before applying the transformation rules, compare every STACK of H_n with f_n, then every STACK of H_{n-1} with $f_n f_{n-1}$, etc. till the STACKs of H_0 are compared with $f_n f_{n-1} \ldots f_0$. If in one of such comparisons the first element is a prefix of the second, terminate the ongoing history.

C2. Terminate the ongoing history if Rule T3 is used.

We now limit our attention to those ongoing histories only that could have appeared in the process of driving, i.e. those which can be constructed starting with a C-ST and applying Rules T1 and T2, before Rule C1 is used.

Lemma 1. If a history segment is not empty then its last stack consists either from one, or from two neighborhoods.

Proof. The lemma is true at the beginning of driving when all history segments are empty. When Rule T1 is used, a STACK which consists of one neighborhood f_n is added at the end of H_n. When Rule T2 is used, H_n disappears, and H_{n-1} gets an addition which ends with $f_n f_{n-1}$.

We shall refer to stacks of length one or two as *short* stacks.

Lemma 2. The situation where one of the history stacks in a segment is a prefix of a later stack in the same or a later segment is impossible.

Proof. Suppose that such a situation exists. Let the earlier stack (to become a prefix) be ab...z, where letters stand for neighborhoods. Each history stack starts at a certain moment when its first neighborhood is the top element of the current stack. The ongoing history at this moment can be seen as:

$$\ldots(\ldots\ ab\ldots z;\ldots\ (H_k\ \ldots(H_n\ f_n)'f_{n-1}\ \ldots)f_k\ )\ldots$$

Here we left out the history segments and context neighborhoods which are common to ab...z and the current stack $f_n f_{n-1} \ldots f_k$, because they only add common endings to both strings. For the earlier stack to be a prefix of the later, f_n must obviously be identical to a. But it is also necessary that f_{n-1} be identical to b. Indeed, f_n can be lengthened only if we open by Rule T2 the internal parentheses marked by the prime '. The use of Rule T1 with any subsequent uses of both rules is irrelevant as long as

the marked parenthesis is not opened (it only creates history stacks subsequent to the stack of interest). This reasoning is also valid for all other elements of the earlier stack up to z; thus we conclude that ab...z must be a prefix of the string $f_n f_{n-1} \ldots f_k$. This, however, is impossible, because Rule C1 (our algorithm of looping back) should have stopped the process at this stage.

Lemma 3 The number of different short stacks is finite.

Proof. The number of different neighborhoods of the first order is finite, because it is the number of paths in a finite tree. Therefore, the number of different stacks of length one or two is also finite.

Theorem. With the driving algorithm described above, no branch of the graph of states may be infinite.

Proof. As shown above, to every branch in the graph, as long as it is not cut off, an ongoing history corresponds. We are now going to show that an infinite ongoing history is impossible.

First we construct yet another model, namely a model of the growth of the ongoing history (which itself is a model of the growth of a branch in driving). The general form of the ongoing history is given by (*). At every stage of the process it consists of a finite number of *levels* separated by parentheses. The part outside of all parentheses is counted as level 0. For $i > 0$, the i-th level is delimited by the i-th and the $i+1$-st nested pair of parentheses, and consists of a history segment H_i and the neighborhood f_i. We want a model which for each level i of the ongoing history will indicate a number of guaranteed short stacks in it. We shall denote this number as G_i. Thus the number of short segments in H_i must be at least G_i. The model describing the dynamics of the numbers G_i is as follows.

At each moment, the highest level n is the level on which an *action* is taken. There are two types of action, which correspond to Rules T1 and T2 above: A1, addition on the level n :

(A1) G_n becomes G_n+1

and A2, cancellation on the level n and addition on the level $n-1$:

(A2) G_n becomes 0, and

G_{n-1} becomes $G_{n-1}+1$, where $n > 0$.

Indeed, when we apply Rule T1 to the ongoing history, a stack of length 1 is added to H_n. When we apply Rule T2, the n-th level disappears, every term in H_n is lengthened by 1 and added to H_{n-1}. We do not know how many short (of length 2) stacks will be there after the operation, and we count it as zero. But one guaranteed stack of length 2 is added to H_{n-1}. After any of the two actions, a new C-ST is created according to both rules, which in our model means that the top level n is incremented by some positive number, and the values of G_i for the new levels are all set to zero.

Suppose now that there is an infinite branch, i.e. an infinite ongoing history. Then the number of levels in it is either limited by a finite number, or infinite. Suppose it is infinite. Some of

the history segments may be empty, others non-empty. We want to prove that if the total number of levels is infinite, the number of levels with non-empty history segments must also be infinite. The total number of empty segments in the ongoing history increases when a new C-ST with at least one new parenthesis is created. Consider separately the cases when the number of parentheses is one, i.e. C-St is (f)g , or more: (...((f)g)...)h. In the former case, the use of Rule T1 transforms one empty segment into a non-empty segment (namely, f;). If Rule T2 is used then the only empty history disappears. In the case of more than one level in the C-ST, one of the empty segments on the level of f or g will be necessarily made non-empty, no matter which of the rules is used. We conclude that the number of empty segments cannot become infinite without making the number of non-empty segments infinite too. Therefore, if the total number of levels is infinite, the number of levels with non-empty history segments will be also infinite.

By Lemma 1 each non-empty history segment H_i ends with a short stack. Since the number of different short stacks is finite (Lemma 3), we must have a situation where two history stacks are identical. This is, however, impossible by Lemma 2.

Therefore, the number of levels must be limited by a finite number, even though the number of actions grows infinitely. Then there must be at least one level i such that an infinite number of actions takes place on that level. The actions, as we know, are of two types: A1 and A2. If the number of actions A2 at the i-th level were finite, then G_i would be infinite, because the number of additions would be infinite while the number of cancellations finite. But this would imply that there are two identical short stacks in H_i, and this is impossible. Therefore, the number of actions A2 must be infinite. However, each action A2 on level i creates an addition on level i-1, hence the number of cancellations, and, therefore, actions A2 on level i-1 must also be infinite. Reasoning in this way we come to the conclusion that the number of actions A2 on level 0 must also be infinite, but this is impossible, because only actions A1 can be performed on that level.

Thus the assumption of an infinite ongoing histoty leads to a contradiciton, which proves the theorem.

R E F E R E N C E S

[1] Turchin, V.F. The concept of a supercompiler. ACM Trans. on Progr. Languages and Systems, vol. 8, No.3, 1986, pp. 292-325.

[2] Turchin, V.F. Program transformation by supercompilation. In Programs as Data Objects, Lecture Note in Comp.Sci. No. 217, pp. 257-325, Springer Verlag, 1985.

[3] Turchin, V.F. The language Refal, the theory of compilation, and metasystem analysis. Courant Institute Rep. #20, New York 1980.