THE SUPERCOMPILER

The supercompiler is a meta-evaluator developed at the City College of New York by the author of the present book with the help of a small group with a varying composition (see historic notes). The concept of metacomputation is very wide and allows different implementations. The supercompiler is one of them. Probably, almost any design decision used here can be replaced by another, and certainly should be critically examined.

6.1 How to generalize?

The central problem of metacomputation 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 theses configurations as nodes. Generalization over configurations is necessary for this. Thus we start this chapter with a description of the algorithm of generalization used in the supercompiler. This algorithm always terminates and produces a finite graph of states and transitions with a self-sufficient set of basic configurations.

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 Refal patterns. 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?

Experimentation with different ways of generalization led the author 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 example, that the strings are scanned from 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 we describe the algorithm of generalization in the supercompiler based on this principle. When we use 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, of course, is 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. In order to define computation histories in detail, we break contractions in the Refal graphs into elementary contractions (see Sec. 3.7).

6.2 Neighborhoods

Definitions. We say that a contraction is *executed positively* over a ground expression, if it is found applicable and applied; we say that a contraction is *executed negatively* if it is established that it 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 the n-th order. Take some computation history of n-th order. The set of all ground expressions with this computation history is a *neighborhood* 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 neighborhood; in other words, the longer the the common history, the *closer* are the points. Note that speaking of ground expressions we have in mind only active ground expressions, i.e. those which include 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.

The reason why we need neighborhoods in metacomputation is exactly that a neighborhood characterized the passage of its member configurations through the working Refal machine. If we know the neighborhood to which a given configuration Q belongs, we know how it will develop during some number of steps to come. If we already have in the current walk a past configuration Q' which belongs to the same neighborhood as Q even though it cannot be reduced to Q, this is an indication that driving may go on infinitely, so we should consider looping back and generalizing Q and Q'.

Let us take a program of a familiar structure:

```
Fab { e1 = <Fab1 ()e1>; }

Fab1 {
  (e2) 'A' e1 = <Fab1 (e2 'B')e1>;
  (e2) s3 e1 = <Fab1 (e2 s3)e1>;
  (e2) = e2; }
```

Its graph, completely factorized, is:

```
{(e0 -> <Fab e1>) (<Fab1 ()e1> <- e0)
+ (e0 -> <Fab1 e1>)
    {(e1 -> s3 e1) {(s3 -> 'A') (<Fab1 (e2 'B')e1> <- e0)
                   +(# s3 -> 'A') (<Fab1 (e2 s3) <- e0)
                   }
    + (e1 -> ) (e2 <- e0)
    }
}
```

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 e1>**. Any call of **Fab1** belongs to a different neighborhood, namely **<Fab1 e1>**. Within this neighborhood we see further differentiations: for example, **<Fab1 ('X')'ABC'>** and **<Fab1 ('PQ')'CC'>** will both come to the next branching point, but then will diverge.

To represent a neighborhood as a pattern we fold up the contractions of the corresponding history. With the graph above, the system of first-order neighborhoods is as follows:

	computation history	pattern expression
(a)	**'f'(Fab)**	**< FAB e1 >**
(b)	**'f'(Fab1)**	**< FAB1 e1 >**
(c)	**'f'(Fab1)B 1 2**	**< FAB1 (e2)e1 >**
(d)	**'f'(Fab1)B 1 2 S 1 3**	**< FAB1 (e2)s3 e1 >**
(e)	**'f'(Fab1)B 1 2 S 1 3 I 3 'A'**	**< FAB1 (e2)'A'e1 >**
(f)	**'f'(Fab1)B 1 2 S 1 3 (# I 3'A')**	**< FAB1 (e2)s3 e1 > (#s3 -> 'A')**
(g)	**'f'(Fab1)X 1 N**	**< FAB1 (e2) >**

These neighborhoods are partially ordered as follows:

a
b > c > d > e
d > f
c > g

where > denotes being a superset.

To compute neighborhoods of the second order, we drive every active walk-end node in the graph for **Fab1**, and thus come to a graph that represents two steps 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 breadth-first can be repeated as long as there are active walk-ends 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*; they 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 exactly identical ways; the machine has never a chance to discover the difference between them.

The idea of the supercompiler is to supervise 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 is necessary. 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 meta-evaluate the configuration

(Q_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 first step of driving replaces Q_1 by the call of **Fab1**, so the graph is the unconditional transition:

$Q_1 = Q_2$
(Q_2) **<Fab1 ()e1>**

Next step of driving results in the graph:

$Q_1 = Q_2$ **{(e1 -> 'A'e1)** $= Q_3$
+ (e1 -> s2 e1) $= Q_4$
+ (e1 ->) $= Q_5$
}

(Q_3) **<Fab1 ('B')e1>**
(Q_4) **<Fab1 (s2)e1>**
(Q_5) *empty*

The passive configuration Q_5 terminates the walk in the graph. None of the new active configurations Q_3 and Q_4 is a subset of any of the previous configurations Q_1 and Q_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

<Fab1 ('BB')e1>
<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 Q_3 and Q_2 are close enough for looping back, or, better to say, not far enough from each other for driving on. 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 Q_3 and Q_2 to *N*, express Q_2 through *N*:

$$Q_2 = \textbf{(empty <- e2)} \,/\, N$$

and recompute the graph for the generalized configuration *N*:

Q_1 (*empty* <- **e2**) *N* {**(e1 -> 'A'e1)** = Q'_3
+ (e1 -> s3 e1) = Q'_4
+ (e1 -> *empty*) Q_5
}

(Q'_3) **< Fab1 (e2'B')e1 >**
(Q'_4) **< Fab1 (e2 s3)e1 >**

Now Q_3 and Q_4 are subsets of *N*; reducing them to *N*, we come to a graph equivalent to the initial definition.

The algorithm of generalization used in the supercompiler 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 configuration that describes the current function call becomes more narrow. Then the replacement is executed, another descending sequence starts, etc. We have the following row of configurations in each branch of the graph (vertical bars mark replacement points):

$$Q_{1,1}\, Q_{1,2} \cdots Q_{1,M1} \mid Q^{2,1}\, Q^{2,2} \ldots Q^{2,M2} \mid \ldots \mid Q_{n,1}\, Q_{n,2} \cdots Q_{n,Mn}$$

They are partially ordered as follows:

$$Q_{i,1} > Q_{i,2} > \ldots > Q_{i,Mi} \qquad \text{for } i = 1,2, \ldots n$$

In a graphic form:

Fig. 6.1 Extension of neighborhoods in a branch of the graph of states

There are several variants of the algorithm, which place the resulting program in different positions on the compilation-interpretation axis. The most interpretive variant is as follows. Each time before we make the replacement of $Q_{n,Mn}$, we compare *now* configurations with *before* configurations. The first *now* configuration is $Q_{n,1}$. The *before* configurations are all the configurations preceding $Q_{n,1}$; they are examined from right to left. The *now* configuration Q^{now} is compared with each *before* configuration Q^{before}. If it is a subset of Q^{before}, we loop back, i.e. do not execute the replacement due, but reduce Q^{now} to Q^{before}. If the two configurations belong to the same neighborhood, we loop back with generalization. In this way we find the tightest generalization for the *now* configuration. If $Q_{n,1}$ does not loop back, we take $Q_{n,2}$ as the *now* configuration, etc. Thus we find the most general configuration that can be looped back. 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.

6.3 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 independent of function definitions. For example, the assignment

```
(<F e1 <G e2> <H e3> > <- e0)
```

will be decomposed into the sequence of assignments:

```
(<G e2> <- eX) (<H e3> <- eY) (<F e1 eX eY> <- e0)
```

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 neigh-

borhoods of S_1 are supersets (generalizations) of the neighborhoods 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 -- this is a definition of a stack. 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 $<F\ E>$, but after executing a number of contractions the Refal machine finds that a not yet computed call in E 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 drive it further. If it is not unitary, it is either passive (completed computation), or non-unitary active (partly 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 drive. If there is no context (the top-of-stack call) and the result of the step is passive, this is the end of driving. If there is no context and the result is only partly passive, the passive part is kept in the view-field of the Refal machine, and the active parts are 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;
      sN = <Mul sN <Fact <Sub sN, 1> > >; }
```

We assume that the arithmetic functions Sub and Mul 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 sequence of operations. We see here three neighborhoods involved:

(f) **<Fact s1>**
(m) **<Mul s1 s2>**
(s) **<Sub s1 s2>**

(To simplify things, we ignore such neighborhoods as **< Fact e1 >**, which in a way goes over in **< Fact s1 >**). 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, of course, includes the special case of an empty *Y*.

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 the developments.

The nested call (*) will now be characterized by the formula *(f)a*. It results from the outside-in driving where we start from the call of **Add**, and 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 other previous stacks. Thus, 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 function **F** scan the argument from left to right and replace 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 meta-evaluate 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 re-appears 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 the 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 > <- e0)

In the short notation,

((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*

Proceeding 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 takes 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 { e1 = <Check <Merge <Pairs e1> > >; };

Merge {
  (e1)(e2)eR = (<Merge2 (e1)(e2)>) <Merge eR>;
  (e1) = (e1);
   = ;}

Check {
  (e1) = e1;
  e1 = <Check <Merge e1> >; }
```

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 e1> >**

where **e1** is a list of pairs.

Driving this configuration outside-in, we have the following row of configurations in the branch where **e1** 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 e1> >**
3. **<C (...)(...) <M e1> >**
4. **<C <M (...)(...) <M e1> > >**
5. **<C (...) <M <M e1> > >**
6. **<C (...) <M (...) <M e1> > >**
7. **<C (...) <M (...)(...) <M e1> > >**
8. **<C (...)(...) <M <M e1> > >**
9. **<C <M (...)(...) <M <M e1> > > >**
10. **<C (...) <M <M <M e1> > > >**

The neighborhoods involved are:

(m)	**< Merge e1 >**
(m_1)	**< Merge (e2) e1 >**
(m_2)	**< Merge (e3)(e2) e1 >**
(c)	**< Check e1 >**
(c1)	**< Check (e2) e1 >**
(c_2)	**< Check (e3)(e2) e1 >**

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 infinity 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. $mc;\ mc_1;\ c_2;\ m_2c;\ mmc_1;\ mm_1c_1;\ m_2c_1;\ c2$

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

6.4 Termination of metacomputation

We now want to prove 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:

e.C-stack = *empty*
| **(e.C-stack) s.F**

Here **s.F** is a neighborhood (function call). In the 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 **e.Stack**:

```
e.Stack = empty
        | e.Stack s.F
```

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

```
e.H = empty
    | e.H e.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, **e.On-hist**, is from the class:

```
e.On-hist = e.H s.F
          | e.H ( e.On-hist ) s.F
```

In a more reviewable form, the ongoing history is:

(*) $H_0(\ldots (H_{n-1}(H_n F_n)F_{n-1}) \ldots)F_0$

here 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 **e.On-hist**. 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 the current branch, or because of looping back to a past stage. After that we shall be able to prove that under those condition no ongoing history, i.e. no branch in the graph, can be infinite.

Below F, H, and S with subscripts stand for neighborhoods, history segments, and past stacks, respectively, and C is a current stack. The starting point of driving is a current stack which represents the initial configuration of the Refal machine. There are three transformation rules for the ongoing history:

T1. *Active replacement rule*:

$H_n F_n$ -> $H_n F_n; C$

T2. *Partly passive replacement rule*:

$H_{n-1}(H_n F_n) F_{n-1}$ -> $H_{n-1} H_n {}^* F_{n-1} F_n F_{n-1}; C$

T3. *Termination rule:*

$$H_n F_n \text{ -> } H_n$$

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 ongoing history 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:

$$H*f = [S_1; S_2 \ldots S_k]*F = S_1F; S_2F; \ldots S_kF;$$

When a step of the Refal machine is performed in the process of driving, one rule must be applied to the ongoing history representing the current branch. The three transformation rules correspond to the three cases in the GMA. 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 results from the step. If the result of the step is passive or non-unitary active, and there is a context (i.e. $n > 0$), 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_nF_{n-1}, is added to the history, and followed by a semicolon; then a new current stack C 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 $n = 0$ (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_nF_{n-1}, etc. till the stacks of H_0 are compared with $F_nF_{n-1}..F_0$. 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 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 of one, or of 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_nF_{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 occurs. 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:

$$...(...\ ab...z;...\ (H_k ...(H_n F_n)'F_{n-1} ...)F_k)...$$

Here we left out the history segments and context neighborhoods which are common to *ab...z* and the current stack $F_nF_{n-1}...F_k$, 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 $F_nF_{n-1}...F_k$. 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) $\quad G_n$ becomes $G_n + 1$

and A2, cancellation on level n and addition on 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 current stack C 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 with at least one new parenthesis is created. Consider separately the cases when the number of parentheses is one, i.e. C 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 current stack C, 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 history leads to a contradiction, which proves the theorem.

6.5 Driving with elementary contractions

As discussed in Chapter 3, one step of driving can be seen as a normalization (an equivalence transformation) of the graph

$$(Q \leftarrow \mathbf{e0})\ G^{\mathrm{P}}$$

where Q is the initial configuration which depends on C-variables, and G^{P} is the graph representing the full program field of the Refal machine and using P-variables. In the course of driving the C-part of the graph grows at the expense of the P-part, until there is no more P-part left, and the step of driving is completed.

In the supercompiler, we use the P-graph G^{P} in the form where contractions are factorized down to the level of elementary contractions (see Sec. 3.7). Factorization of contractions requires some modification in the algorithm of outside-in driving described in Sec. 3.5. We shall explain this on the following example.

Consider this clash:

(< F < G e1 > B > <- e0) (e0 -> < F s2 e1 A >)

where the part coming from the P-graph is not factorized. This requires the matching

< G e1 > B : s2 e1 A

The algorithm of outside-in driving which we described in Sec. 3.5 will work as follows. We first try to project **s2**, but see that on the left edge of the argument there is a hindrance. Then we switch to the other edge, and see that the matching is impossible. We have spared the effort of driving the hindrance **< G e1 >**.

If the P-graph is factorized into elementary contractions, the same clash will be:

(< F < G e1 > B > <- e0) (e0 -> < F e1 >)(e1 -> s2 e1)(e1 -> e1 A)

Each elementary contraction describes one act of projecting an element of the pattern on the argument. When we factorize a contraction, we fix the order of projecting. Thus we come to the matching:

< G e1 > B : s2 e1

which is aborted because of a hindrance. If we make the hindrance **< G e1 >** the next primary active expression, we shall do unnecessary work on its driving, because in the end all the branches starting from this point will be cut off on the account of the next contraction. The supercompiler will go on with 'conditional driving'. It will examine further contractions and execute those which can be 'easily' (we do not want to go into details) executed, even though some of the previous contractions stopped were because of the hindrance. If there was no 'matching impossible', it will return to the hindrance and drive it. In our case **(e1 <- e1 A)** will be executed, and the supercompiler will discover, without driving the hindrance, that the matching is impossible.

This algorithm is somewhat messy; the factorization of contractions certainly does not help, and this can be seen as a disadvantage of factorization. The advantages, however, are more significant.

We can point at two advantages of reducing graphs to elementary contractions. The first is the simple structure of paths in the graphs. This creates the possibility of to various efficient codings and helps in writing algorithms. The second, and the most important, advantage is the simplification of operations on neighborhoods. As we drive a function call, say, $<F\ E>$, against the graph of F, the neighborhood to which each next contraction of $<F\ E>$ belongs is uniquely defined by the path from the root of the P-graph for F to the current point. In the supercompiler, we identify neighborhoods with these paths (in the three-symbol code). Thus when we make each step of the normalization involving an elementary contraction, we simply concatenate it to the current P-path. If we kept neighborhoods in their natural form as L-expressions, we would have to make a composition of contractions, which requires a substitution, and this is a costly operation. To establish that one neighborhood is a subset of another, we check that the latter is a prefix (the beginning part) of the former. With L-expressions, we would have to use the GMA.

Let us consider the driving with a neighborhood in greater detail. The initial stage is:

(1) $\quad (<F\ E> \leftarrow \mathbf{e0})\ G^{\mathrm{P}}$

Only the subgraph of G^{P} for F is used, which is

$$(\mathbf{e0} \rightarrow <F\ \mathbf{e1}>)\ G^{\mathrm{P}}[F]$$

where $G^{\mathrm{P}}[F]$ is the graph for F. The first step of driving by normalization is the resolution of the clash for **e0** results in stage number two:

(2) $\quad (<F\ \mathbf{e1}> \leftarrow \mathbf{e1})\ (E \leftarrow \mathbf{e1})\ G^{\mathrm{P}}[F]$

Suppose that G^{P} starts with the contraction (**e1 -> s2 e1**). Then at the next step we have the clash for **e1**:

$$(<F\ \mathbf{e1}> \leftarrow \mathbf{e1})\ (E \leftarrow \mathbf{e1})\ (\mathbf{e1} \rightarrow \mathbf{s2\ e1})\ G^{\mathrm{P}}[F(\mathbf{e1} \rightarrow \mathbf{s2\ e1})]$$

The part of the P-graph which is still there is what remains after we use the first two elementary contractions. The resolution of the clash required here may give zero, one, or two branches of the form:

(3) $\quad C_1\ (<F\ \mathbf{e1}> \leftarrow \mathbf{e1})\ (\ldots \leftarrow \mathbf{e1})\ (\ldots \leftarrow \mathbf{s2})\ G^{\mathrm{P}}[F(\mathbf{e1} \rightarrow \mathbf{s2\ e1})]$

where C_1 is the C-contraction (for this branch) resulting from the clash resolution, and we have put dots for some values of the P-variables **e1** and **s2**.

Let us now look at the sequence of neighborhoods to which the current configuration belongs. The following table shows the three initial stages of the neighborhood, the corresponding path, and the code for the path we use in the supercompiler:

Stage	Neighborhood	Path	Code of Path
1	**e0**	*empty*	*empty*
2	**<*F* e1>**	**(e0 -> <*F* e1>)**	**'f'(*F*)**
3	**<*F* s2 e1>**	**(e0 -> <*F* e1>)(e1 -> s2 e1)**	**'f'(*F*)'S'1 2**

The configuration is defined by the neighborhood to which it belongs and the assignments for the P-variables of the neighborhood. Suppose that the initial configuration is **<*F* 'AB' e4>**. Then the C-to-P assignments and the corresponding configurations for the initial stages of driving are:

Stage	C-to-P assignments	Configuration
1	**(<*F* 'AB' e4> <- e0)**	**<*F* 'AB' e4>**
2	**('AB' e4 <- e1)**	**<*F* 'AB' e4>**
3	**('B' e4 <- e1)('A' <- s2)**	**<*F* 'AB' e4>**

The configuration is the same at all three stages of driving in this case, because it passed through them without contractions. It is only its representation that has changed. If we take **<*F* e4'B'>** as the original configuration, it will split into two configuration at stage 3. They will still belong to the same neighborhood as above but for one branch the contraction C_1 in (3) will be **(e4 ->)**, the configuration **<*F* 'B'>**, and the C-to-P assignments

(<- e1)('B'<- s2)

while for the other branch C_1 will be **(e4 -> s5 e4)**, the configuration **<*F* s5 e4'B'>**, and the assignments

(e4'B' <- e1)(s5 <- s2)

Thus when we 'drive with a neighborhood', the current configuration Q is split into a neighborhood N and its specialization by a C-to-P assignments:

$$Q = (E \leftarrow \underline{\mathrm{var}}(N))/N$$

This representation of Q characterizes the current *situation* (in driving). We also include in the situation the current restrictions on the C-variables $\underline{\mathrm{var}}(Q)$, and a whole number **s.Frind** which gives us the first index which is free to use for the index of the next *new* variable (each time we introduce a new variable, **s.Frind** is increased by one). With these additions the *situation* includes all information we need to go on with driving in the current branch.

In Sec. 3.6.3 we gave the scheme of driving by normalization. The input to the procedure is the assignment to **e0** of the initial restricted configuration, which has the formula $R^c A^{c0}$, and a path (not factorized) in the P-graph rep-

resenting one Refal sentence, with the formula $C^{0p}R^{p}A^{p0}$. If we split the contraction in the P-graph in two parts: from **e0** to the p'-set, and from this set to p, then we have the formula:

$$C^{0p'}R^{p'}C^{p'p}R^{p}A^{p0}$$

We shall understand the p'-set as the variables defined in the current point of the P-graph, which moves forward as we drive on.

The procedure of driving starts with the concatenation of the C-piece and the P-piece:

$$R^{c}A^{c0}C^{0p'}R^{p'}C^{p'p}R^{p}A^{p0}$$

After the normalization of the part $R^{c}A^{c0}C^{0p'}R^{p'}$, i.e. after traveling from the root of the P-graph to the current point, we have the picture:

$$C^{cc'}(R^{c'}A^{c'p'})C^{p'p}R^{p}A^{p0}$$

Here the parenthesized part represents what we called above the current *situation*. It stands at the juncture of the C-part and the P-part. On the left of the situation is the ready part of the graph. The contractions of this part transform the original set of C-variables into the current derivative set C'. On the right of the situation we have the part of the original P-graph that did not yet take part in normalization. Its current set of variables is P'. We must expect further contractions and restrictions for P'-variables, until we come to the final assignment. The situation, as it is seen in this formula, includes only the restrictions for the current C-variables C', and the C-to-P assignments. There is no neighborhood. Indeed, we do not need the neighborhood for further driving. We keep it in the situation because it is needed for other phases of metacomputation.

6.6 P-graphs and C-graphs

We are now in a position to define the basic formats we use to represent metacoded Refal graphs as Refal objects for the supercompiler. In the general structure of objects we closely follow the definition of Refal graphs in Sec. 3.6.1; variations serve the purposes of efficiency. But the whole object we deal with in driving is a combination of P-graphs and C-graphs, and it also must keep the history of computation. Therefore it is much more complex.

The graph we keep in the computer in the course of metacomputation can be seen as a three-dimensional structure. The objects we deal with are structured in space by structure and activation brackets; thus *space* is the first dimension. We trace the development of these objects in time, and record its history -- the second, *time*, dimension. We have free variables, and for different cases of the values of these variables the development in space and time is different. This is the third, the *case* dimension. This three-dimensional object must me mapped on a Refal expression, which is a one-dimensional object.

6.6.1 CONTRACTIONS, RESTRICTIONS, ASSIGNMENTS

In the formulas below, the indexes of Refal variables represent various syntax types of objects. Different representatives of the same type are distinguished by adding numbers to their names. The bar | (to read 'or') separates different variants of a syntax type. Some syntax types are designated with the use of **u** as a variable type. Of course, there are no such variables in Refal. They are used only in syntax definitions. A u-variable (from 'Unit') stands for a structure which has a definite (but different for different variables) number of *ultimate terms*. For instance, **u.contraction** consists of exactly three symbols (see below). When we design structures for a Refal program, we try to define as many of the needed structures s- or t- or u-variables, so as to be able to recognize them efficiently when we scan the expression from left to right or from right to left. e-variables should be used only when the structure is a sequence of an arbitrary number of basic units. Of course, we can do without u-structures by using one more pair of parentheses. A contraction, for example, could be represented by a t-structure (sT sI s3). But we can equally well do without the spare parentheses.

u.Contraction = s.Contr-type s.Var-index s3

t.Restriction = ('#' e.Contr-graph)

e.Contr-graph = u.Contraction e.Contr-list
| e.Contr-list ':'(e.Contr-gr-sum)

e.Contr-gr-sum = (e.Contr-gr)
| (e.Contr-gr) '+' e.Contr-gr-sum

e.Contr-list = *empty*
| u.Contraction e.Contr-list

e.Assignment = (e.Expression) s.Var-type s.Var-index

e.Assign-list = *empty*
| e.Assignment e.Assign-list

Examples: the contraction **(e3 -> e3(e4))** is represented as **'C' 3 4**. The assignment

((X1 '*' X2) '+'s.Gone eX s7) <- e.105)

becomes

((X1 '*V' X2)'+*S' Gone '*E'X '*S'7)'E'105

Syntax:

```
e.P-Graph = e.P-Segment t.P-Walk-end
          | e.P-Segment ':'(e.P-Graph-sum)

e.P-Graph-sum = (e.P-Graph)
          | (e.P-Graph) '+' e.P-Graph-sum

e.P-Segment = e.Contr-list
          | e.Contr-list t.Restriction

t.P-Walk-end = ('=' e.Expression)
```

Thus restrictions in a P-graph can be placed only immediately before branchings or walk-ends. The expressions in walk-ends are the right sides of sentences.

In the interpretation of a P-graph, the sentences of the Refal program, which are now the branches of the graph, are tried consecutively until one that is applicable is found. This dictates a certain interpretation of the nodes in the P-graph which result from factorization.

Consider the graph

$$\{\, C_1\, C_2\, W_1 \;+\; C_1\, C_3\, W_2 \;+\; C_4\, W_3 \,\}$$

where C_i are some contraction segments, and W_i some walk-ends. After factorization it becomes

$$\{\, C_1\, \{\, C_2\, W_1 \;+\; C_3\, W_2 \,\} \;+\; C_4\, W_3 \,\}$$

This graph is presented in Fig.6.1, where we have denoted the two nodes in the graph as Q_1 and Q_2. The node Q_2 appears because of factorization.

Fig. 6.1 A P-graph with a factorization node

How should this graph be interpreted? Suppose C_1 is found applicable, then C_2 inapplicable. We proceed to the second branch starting at Q_2, after restoring the situation as it became after C_1, i.e just before entering the node. If C_3 is applicable, we take up the walk-end W_2, and this is the end of the step. If C_3 inapplicable, we do not yet declare an abnormal stop, but go back beyond Q_2, restore the situation as it was at the point Q_1, and try C_4, in order not to miss the third sentence in the original definition.

This interpretation is different from the interpretation of a graph of states where the nodes represent configurations resulting from completed steps of the Refal machine. Indeed, suppose that the graph in Fig... is a graph of states (e.g., a C-graph resulting from metacomputation) where Q_2 marks the configuration Q_2 resulting under condition C_1 from one step of the Refal machine with the preceding configuration Q_1. In the interpretation of this graph, if C_2 and C_3 are inapplicable, we must not go back beyond Q_2, but declare an abnormal stop.

The difference between the interpretation of factorization nodes and configuration nodes creates a problem. A factorization node in the P-graph must go over into a node in the C-graph which also should be interpreted as a factorization, not a configuration, node. This can be best seen from the special case when the argument of the current function is simply a free variable; the C-graph then replicates the P-graph. Thus we have to maintain a difference between two kinds of branchings, or nodes: configuration, no-return nodes, and factorization nodes, which must be passed around in interpretation. This is inconvenient in many respects. It is especially unpleasant that the semantics of factorization nodes complicates the concept of the history of computation. Indeed, if we passed through a factorization node, it does not yet mean that this node will be present in the final history of computation; it may still be passed on the way back, and disappear.

To avoid these complications, we put a restriction on factorization which allows only such nodes that do not require passing them in backward direction; these nodes, are essentially configuration (functional) nodes -- they break the domain of the function into disjoint subfunction domains.

Examples. Let the graph produced directly from Refal sentences be

```
{ (e1 -> 'AB'e1) Q1
+ (e1 -> 'AC'e1) Q2
+ (e1 -> 'BCB'e1) Q3
}
```

It can be factorized in the obvious way:

```
{ (e1 -> 'A'e1) { e1 -> 'B'e1) Q1
                + (e1 -> 'C'e1) Q2
                }
+ (e1 -> 'BCB'e1) Q3
}
```

Because **(e1 -> 'A'e1)** is orthogonal to **(e1 -> 'BCB'e1)**, the factorization node which has appeared in the graph is a no-return node, thus this factorization is legitimate. Consider, however, this modification of the graph:

```
{ (e1 -> 'AB'e1) Q1
+(e1 -> 'AC'e1) Q2
+(e1 -> s2 e1 s2) Q3
}
```

Now the first elementary contraction chopping 'A' cannot be factorized. Indeed, if we try to do this, we have

```
{ (e1 -> 'A'e1) { ( (e1 -> 'B'e1) Q1
                  + (e1 -> 'C'e1) Q2
                  }
+ (e1 -> s2 e1 s2) Q3
}
```

which includes a prohibited factorization node. If the argument starts with **'A'**, this does not yet mean -- against our best habits -- that the first branch will be taken. If the argument is **'ADA'**, we pass the first branch, enter the node, try both branches in the subgraph, fail, and --in violation of our agreement -- must return to the opening node to find finally that the last sentence is applicable. This situation cannot be corrected by adding restrictions. If the restriction on a branch is produced by subracting only the preceding parallel *segments*, not full branches, then the algorithm will be altered. In our example, if on the last branch we put the restriction **(# e1 -> 'A'e1)**, then the function becomes undefined on such arguments as **'ADA'**.

The requirement we set for a P-graph used in metacomputation is as follows. In any subgraph of the P-graph, consider a segment W_2 which leads to a branching point:

$$\{G_1 + W_2\,\{G_2\} + G_3\}$$

Then W_2 must be orthogonal to all branches in G_3. If this condition is met, the node at the root of G_2 can be treated as a no-return node.

The no-return limitation on factorization is rather strong. Sometimes it will demand that we abandon a very useful factorization. In such cases it may be worthwhile to restructure the Refal definition giving priority to the desired factorization, and agreeing to pay some price for it in terms of the size of the program. Consider the following variation of the example above. Let the Refal program be:

```
F1 {
  'ABCDEFGHIJKX' e1 = Q1;
  'ABCDEFGHIJKY' e1 = Q2;
  'ABCDEFGHIJKZ' e1 = Q3;
  s2 e1 s2 = Q4; }
```

In this definition we cannot factorize **(e1 -> 'ABCDEFGHIJK' e1)** without violating the no-return requirement. But it would be a shame to check several times that the argument starts, or does not start, with the letters from A to K. We can incorporate this check into the algorithm so that it is performed only once:

```
F1 {
  'ABCDEFGHIJK' e1
          with {e1 : 'X' e1  =  Q1;
                e1 : 'Y' e1  =  Q2;
                e1 : 'Z' e1  =  Q3;
                e1 : e1 'A'  =  (('A')('BCDEFGHIJK') <- (s2)(e1))/Q4;
               };
  s2 e1 s2  =  Q4;
  }
```

The price we had to pay for this is one extra sentence. But in terms of execution time this program, when it is transformed into a Refal graph and interpreted, is fully efficient.

Exercise ... Derive formally the second definition from the first.

Another situation where the no-return requirement may prevent a useful factorization is an overgeneralized 'catch-all' sentence. Consider a function which has the format **< F (e1)(e2)e3 >**. Suppose that two sentences:

$$\langle \mathbf{F}\ (A)(B)C \rangle = R_1$$
$$\langle \mathbf{F}\ (D)(E)F \rangle = R_2$$

where A, B, etc. are some L-expressions, describe two cases we want to distinguish. And then, suppose, we want to add that otherwise something else should be done, like sending an error message or stopping, which does not require an analysis of the arguments. We can simply write:

$$\langle \mathbf{F\ e1} \rangle = R_3$$

But then it will be impossible to to take out the format of the first two sentences as the factor

(e1 -> (e2)(e3)e1)

To be able to do this factorization, we must rewrite the third sentence using the format, i.e.

$$\langle \mathbf{F\ (e2)(e3)e1} \rangle = R_3$$

Unfortunately, this kind of transformation can hardly be made automatically, because when we use the format in the last sentence we limit the domain where our function is defined. If a program makes this transformation, it will not know whether the human programmer simply neglected to write out the format, or he wanted the definition to work also in the case when the format of the preceding sentences is violated.

One part of the supercompiler is a program which transforms a Refal program into a P-graph with no-return nodes. Thus all nodes in the C-graph produced by the supercompiler must be interpreted as no-return nodes too.

Syntax:

```
e.C-Graph  =  e.C-segment e.Final-C-assign
           |  e.C-segment ':'(e.C-Graph-sum)

e.C-segment  =  empty
           |  u.Contraction e.C-segment
           |  u.C-func-assign e.C-segment
           |  u.C-graph-assign e.C-segment

u.Final-C-assign  =  ('C' e.Passive-expression)'E'0
           |  ('F'e.Func-name e.Assign-list)'E'0

u.C-func-assign  =  ('F'e.Func-name e.Assign-list)'E's.Var

u.C-graph-assign  =  'G'(e.C-graph)'E's.Var

e.C-Graph-sum  =  (e.C-Graph)
           |  (e.C-Graph) '+' e.C-Graph-sum
```

There are no restrictions in C-graphs (they are implicit). The function calls in function-call assignments **u.C-func-assign** and final assignments **u.Final-C-assign** represent basic configurations. The arguments of these functions are the free variables in the corresponding configurations. The assignment lists assign values to these variables. Function names are of two kinds. First, it may be the name of a predefined basic configuration, in particular, a built-in function. This name will usually be an identifier. Second, it may be a configuration declared as basic in the course of metacomputation. This will be a whole number. The initial configuration has 1 as its number.

There are also graph assignments **u.C-graph-assign** in C-graphs. To execute such an assignment we execute the graph **e.C-graph** and reassign the final value of **e0** to the e.variable with the index **s.Var**. While in P-graphs the walk-ends are simply right sides of Refal sentences, the structure of a C-graph reflects decompositions with the separation of subgraphs, which the supercompiler performs in the process of computation. A walk-end in a C-graph is the final stage of a walk-end in a CP-graph, which we proceed to discuss.

6.7 CP-graphs. History records

The overall data structure with which the supercompiler works is a *CP-graph*. It includes P-graphs, C-graphs, and *situations* as interfaces between them. The supercompiler uses the multibracket technique to pass the CP-graph from left to right, so that at the pointer we have the current C-to-P interface, and at the left from it there is the C-part, and at the right the P-part of the graph. As it moves through the CP-graph, it transforms P-parts into C-parts.

There are two major differences between the C-part of the CP-graph and a finished C-graph. First, a CP-graph is a structure of parallel and nested *subgraphs* of several types (a C-graph can include only C-graph- assignments).

Second, a CP-graph includes *history records*, which are necessary in order to compare the current node with the past nodes and loop back at appropriate moments. History records may also include the information about the basic situations which are declared at the moment.

First we discuss history records. There are two types of those: *nodes* and *situations*.

t.Node = ('N'e.Node) = ('N' s.Frind (e.Contr-graph) e.Expression)

A node represents a configuration of the Refal machine in its "natural", i,e, not decomposed into a stack of function calls, form. The expression in the node is an arbitrary general Refal expression; it may have any member of levels of nested function calls. The contraction graph represents the restriction **('#'e.Contr-graph)** on the variables of the expression. The empty **e.Contr-graph** signifies the absence of restrictions.

We keep nodes in the history of metacomputation in order to be able to loop back to a node if we meet it, or its specialization, again. However, from the examples in the beginning of this chapter we know that if we always waited till a node repeats itself in order to loop back, we would almost certainly never finish the construction of the graph. This is why we also keep situations as history records.

The syntax of situations is:

e.Situation = s.Frind (e.Contr-graph) e.Func-call e.Context

t.Func-call = ((e.Path) e.Assign-list)

e.Path = 'f'(e.Func-name) e.Contr-list

e.Context = e.Stack

e.Stack = *empty*
| **t.Func-call e.Stack**

Function calls are represented as neighborhoods specialized by assignment lists, as discussed above. The *path* in a function call can be seen as a new function name. This is a function which is defined only on a subset of the original function **e.Func-name**, but in this subset it is identical to **e.Func-name**. A *stack* is defined in Sec.... In the decomposition of nested function calls into stacks we always use the same standard variable as a liaison between function calls in the stack. It has a special type indicator **h** and the character **'X'** as the index: **h.X**. In the metacode it is **'*HX'**. The function call on the top of the stack in a situation determines the further development of the situation at the moment. The further part of the stack is a context. There is a difference between the function call in the situation and the function calls that constitute the context. The former does not include h-variables, while each of the latter has exactly one entry of **h.X**.

The general syntax of CP-graphs is:

e.CP-graph = e.Contr-list e.CP-walk-end
| **e.Contr-list e.Ready-grs e.Reduction u.Branching**

```
u.Branching = u.Fork-situation ':'(e.CP-graph-sum)
            | u.Past-situation ':'(e.Cp-graph-sum)

e.CP-graph-sum = e.C-begin (e.CP-graph) e.P-end

e.C-begin = empty
          | (e.C-graph)'+' e.C-begin

e.P-end = empty
        | '+'(e.P-graph) e.P-end

u.Fork-situation = '$'(e.Situation)

u.Past-situation = (e.Basics-list)'P$'(e.Situation)
```

A situation is found in a CP-graph in front of every branching point, and at the interface between the C-part and the P-part. The role of the interface situation was described in Sec.6.5 (old). The situation at a branching point represents the configuration at that point. We need it in order to jump to the next branch of the P-graph when the current branch comes to an end. We distinguish between *Fork situations* and *Past situations*. The former are kept only for jumping to parallel P-branches, as we have just described. The latter have an additional role as possible points to loop back in the process of metacomputation. We never try to loop back on a fork situation.

It should be noted that past situations can appear only in front of branching points. This reflects our strategy of dealing with transitory situations. If a configuration goes over into another configuration without contractions, there will no branching point, and therefore, no **e.Past-situation**. The history of computation by unique steps is not recorded. From this rule, however, as from any rule, there may be exceptions. In some situations we still want to keep a history record of a configuration, even though it is transitory. Then we create for it a special fictitious branching point which consists of only one branch. Thus the syntax is not violated.

In order to understand why we have to do this distinction, we must recall that we loop back only to a configuration which is separated from the current configuration by at least one step of the Refal machine. If two situations are separated only by contractions, the later is only a specialization of the earlier, and to loop back would be absurd. However, when we meet a factorization node in the current P-graph, we have to remember the current situation, and thus create a branching point with a Fork situation. This is a situation separated only by contractions from the next situation up the tree, so we do not want to loop back. An analogous situation can appear because of outside-in driving, even without factorization nodes. In outside-in driving it happens that after we have left a situation at a branching point Q_1 and made a contraction, we meet a hindrance at a point Q_2, and have to decompose the configuration and use the P-graph for another function. Again, we cannot loop back from Q_2 to Q_1. Generally, we have a number of consecutive situations which belong to the same step of the Refal machine. At a certain moment one, or some, of them must be chosen as Past situations, while the other remain Fork-situations. We do not go here into the details of the algorithm used by the supercompiler.

The situation at the C-to-P interface is a part of **e.CP-graph** in **e.CP-gr-sum**. The CP-graph **e.CP-graph** can, of course, contain **e.CP-gr-sum**. This recursion can take place any number of times, but in the end **e.CP-graph** will end with a walk-end **e.CP-walk-end**. This data structure will be described in Sec. 6.7 and 6.9. One of the variants of **e.CP-walk-end** is a C-to-P interface.

An example of an actual computer output will help to see the data structures used by the supercompiler. We take a very simple problem. The definitions in the program field are:

```
Fa {
  'A'e1 = 'B'<Fa e1>;
  s2 e1 = s2 <Fa e1>;
   = ;
   };

* Iterative Fb
Fbi { e1 = <Fbi1 ()e1>;};
Fbi1 {
  (e1)'B'e2 = <Fbi1 (e1'C')e2>;
  (e1)s2 e3 = <Fbi1 (e1s2)e3>;
  (e1) = e1;
     };

Fc {
  'C'e1 = 'D'<Fc e1>;
  s2 e1 = s2 <Fc e1>;
   = ;
   };
```

The function **Prout**, which is used by the supercompiler for output, is primarily oriented at printing strings of characters; it prints such a string exactly as it is in the argument, without any quotes. Refal does not set any particular way of printing expressions; this is left to the user. However, if there are non-character symbols and parentheses in the argument of **Prout**, it will print them out in a certain manner, namely, parentheses are printed as parentheses, identifiers and numbers are printed as the corresponding strings of characters separated from adjacent symbols by blanks. Obviously, this must be used only when there is no risk of confusing parentheses with character-parentheses, and non-character symbols like the number **25** with their character representations like **' 25 '**. Since the user, as a rule, knows what to expect from the output, the simple use of **Prout** usually is satisfactory. Below we shall list object expressions as printed by **Prout.**

The P-graph for **Fa** printed in this way is:

```
:((S 1 2 :((I 2 A(=B*((Fa )*E 1 )))
 +((#I 2 A)(=*S 2 *((Fa )*E 1 )))))
 +(X 1 N(=))
 )
```

Let us give the following initial configuration to the supercompiler:

< Fc < Fbi < Fa e1 > > >

The initial node is:

('N' 2 ()'*'((Fc)'*'((Fbi)'*'((Fa)'*E'1))))

Printed by the computer, it is

```
(N 100 ()*((Fc )*((Fbi )*((Fa )*E 1 ))))
```

We leave the numbers from 1 to 99 for the variables in the initial configuration, and start the indexes of new variables from 100. The initial situation, which represents the same configuration as the node, is:

```
$(100 ()((f(Fc ))(*((Fbi )*((Fa )*E 1 ))E 1 ))
```

Note that at the beginning the the stack consists of one function call only -- the outermost one. The breaking down of this nested call will be done in the process of outside-in driving.

The reader, hopefully, has already had enough exposure to metacomputation in order to see what will the supercompiler do with this task. Since **Fbi** calls **Fbi1**, and the latter calls itself recursively, without putting anything out of the activation brackets, there is no choice but to declare

(1) **< Fbi1 < Fa e1 > >**

a basic configuration. The two-pass processing represented by this nested call will be replaced by a new one-pass function (namely **F2**, except that **F2** will include **Fc** as a context, as discussed below). The function **Fc** will remain as it is, except that it is renamed as **F3**. Thus the three-pass algorithm will be transformed into a two-pass algorithm.

This is the final output of the supercompiler:

```
Recurrent Basic #3 :
$(103 ()((f(Fc ))(E 102 )E 1 ))

Graph:
:((
   S 102 103
   :((
      I 103 C(F 3 (*E 102 )E 102 )E 104 (CD*E 104 )E 0
     )+(
      (F 3 (*E 102 )E 102 )E 104 (C*S 103 *E 104 )E 0
    ))
  )+(
   X 102 N(C)E 0
 ))
====================

Recurrent Basic #2 :
$(103 ()
  ((f(Fa ))(*E 1 )E 1 )
  ((f(Fbi1 )B 1 2 )(*E 102 )E 2 (*HX)E 1 )
  ((f(Fc ))(*HX)E 1 )
 )

Graph:
:((
   S 1 103
   :((
      I 103 A(F 2 (*E 1 )E 1 (*E 102 C)E 102 )E 0
     )+(
      I 103 B(F 2 (*E 1 )E 1 (*E 102 C)E 102 )E 0
     )+(
```

```
      (F 2 (*E 1 )E 1 (*E 102 *S 103 )E 102 )E 0
    ))
  )+(
   X 1 N(F 3 (*E 102 )E 102 )E 0
))
====================

Graph for the initial configuration:
(F 2 (*E 1 )E 1 ()E 102 )E0
```

Both basic configurations emerged from looping back to a situation, not to a node. Thus they are defined in terms of situations. The basic **F2** has three calls in the stack, not two as one could expect. Even though it is the two-call configuration (1) that necessitates looping back, the basic configuration, as ultimately defined by the supercompiler, incorporates the context **Fc**. As a result, the final graph does not use the nested call **< F3 < F2 ... > >**, but simply calls **F2**, which calls **F3** when it completes its part of the job.

6.8 Local lists of basic configurations

When we generalize Q^{after} with Q^{before}, we return to the point Q^{before}, delete the whole development of this configuration, reduce it to the generalization, and go on from this point. This situation is represented schematically in Fig. 6.2, where solid lines stand for Refal steps, and dashed lines show reductions. Configuration $Q1$ produces Q_2, which is of the same neighborhood Q_n as Q_1, but cannot be reduced to it, i.e. is not its subset. So we must form a generalization, in accordance with our algorithm. We can take the full neighborhood Q_n as the generalization. But it may be possible to make a tighter generalization, and the supercompiler uses this possibility. Those variables of Q_n for which the values in Q_1 and Q_2 are identical are retaining these values; the variables which have different values are generalized to their syntactic types, i.e. *sI* or *eI*.

Fig. 6.2 Repeated generalization

Let the generalization of Q_1 and Q_2 be Q_g. The whole previous development of Q_1, including Q_2, is now deleted, and Q_1 is reduced to Q_g (FIg. 6.2a). We develop Q_g now. Suppose that, as shown in Fig. 6.2b, the first branch exiting from Q_g, which produced Q_2 before, produces Q'_2 recognizable as Q_g. So this part of development is successfully completed. The other branch, however, produces Q_3, which belongs to the same neighborhood Q_n, but can-

not, again, be reduced to Q_g, so it requires a next generalization. Therefore, Q_g does not stay in the final graph as a basic configuration. If we included it in the list of configurations declared basic, it must be excluded from that list at this stage, reduced to the new generalization (that of Q_3 and Q_g), and redeveloped.

This is, however, only one part of the problem. Consider the situation in Fig.3. A basic configuration Q_1 produces configuration $Q2$, which loops on itself, and is thus declared basic. The development of Q_2 includes a call of Q'_1, which is reduced to Q_1. There is only one more branch of Q_2, which happens to lead to a passive node P. Now the development of Q_2 is successfully completed; we have not only a basic configuration, but also completed development, which we may output as a part of the final program.

Fig. 6.3 Generalization of a parent configuration

Then we go on developing Q_1, and discover that it produces Q_3, which demands generalization of Q_1. We have to throw away the whole development of Q_1, reduce it to its generalization with Q_3, and redevelop it. However, we have in the output the basic configuration Q_2, which calls the now nonexistent basic configuration Q_1, for which we shall never have a graph. So, not only must we delete Q_2 from the list of basic configurations, but we must also delete its definition from the output. In fact, we should not have sent the graph for Q_2 to the output, in the first place.

These problems are solved by making basic configurations local to those nodes of the graph on the existence of which they may depend. Instead of maintaining one global list of basic configurations, we keep local lists at the branching points leading to the current configuration; see the variable **e.Basic-list** in **u.Past-situation**. For each basic configuration in the list we keep: its index (a whole number) and its definition in C-terms, which is a restricted configuration. We might also keep the configuration's development (a graph for it), but it could take too much space in the memory. Therefore, we do not keep it in the list, but put on disc as part of a separate list of configuration graphs. At the end of metacomputation this list is read, and the graphs for the disappeared basic configurations are removed.

The local lists of basics are maintained in the following way. When a Past situation is looped at, we declare it a *tentative basic*.

As we develop a tentative basic, one of three things can happen: (1) it is generalized again with some configuration in its development and redeveloped, still remaining a tentative basic; (2) it disappears because some node up the tree is generalized and redeveloped; (3) its development is successfully finished. In the third case the tentative basic becomes a (real) basic

and is added to the list stored at the next Past situation up the tree. In this way every configuration declared basic at any branch will become available for all following branches. When we check whether the current configuration is basic, we consult all Basic lists up to the root of the whole CP-graph.

Those basic configurations that originate from nodes, not situations, are never generalized. When They are simply added to the closest **e.Basic-list** when we move up the graph in the process of depth-first scan of the tree.(See the next section for the syntax of a node used as an historic record).

There are a few more decisions a meta-evaluator must take when running through the lists of basic configurations. If the current configuration Q and some basic configuration Q^b belong to different neighborhoods, then we certainly can ignore Q^b and go over to the next basic configuration. If Q is a subset of Q^b then, again, it is clear that the reduction is in order. Suppose, however that Q belongs to the same neighborhood that Q^b, but is not its subset. Then two courses of action are possible.

(1) We can generalize Q and Q^b, and substitute the generalization for Q_b in the Basic list. We must then reduce Q to the generalization and go on with metacomputation. The inconvenience of this method is that we also have to look through the developments of all basic configurations and replace the calls of Q^b by equivalent calls of the more general basic.

(2) We can ignore Q^b, and this is what the supercompiler does. With this method, Q may become a new basic configuration which belongs to the same neighborhood as Q, but may be in different relations to Q^b: disjoint, partially overlapping, or covering (more general).

The choice of the second method raises other questions. Since the lists of basic configurations may include overlapping configurations, the order in which we examine them may be of significance. Suppose we have two basic configurations Q^{b1} and Q^{b2}, such that the former is a subset of the latter. Then we must first compare Q with Q^{b1}, and only then, if the reduction is impossible, with Q^{b2}, otherwise Q^{b2} will never be used. But to use a tighter basic configuration is always desirable; as we know, this can give a gain in efficiency. Since more general basics come later than their subsets (a subset cannot come later because it will be recognized as belonging to an already existing basic configuration), we must compare the current configuration with basic configurations in the same order in which they are added to the lists. As far as different lists go, this means that we should use the Basic lists starting with root of the CP-graph and ending with the current node. This order is opposite to that highly desirable for the efficiency of looping back. Indeed, algorithms often have the structure of nested loops. Thus we have better chances to loop back (either to a basic configuration, or to a Past situation) if we examine the nodes of the CP-graph in the direction from the current configuration to the root. To satisfy both requirements we have to go through the graph twice: first from the root to the node, and then back to the root. This also is undesirable.

The supercompiler makes a compromise decision. It passes the current branch only once, and in the direction from the node to the root, comparing the current configuration both with Past situations, and with basic configurations. Inside a local list, however, the scan is in the same direction (left to right) as the direction in which the list grows when new basics are added.

6.9 CP-graphs. Subgraph structure

Syntax:

e.CP-walk-end = u.Node-context
| e.Interface
| u.Past-node
| u.X-subgraph
| u.Null-subgraph
| e.Subgraph-list ('A' u.Final-C-assign)
| e.Subgraph-list ('A' u.Final-C-assign) '$'(e.Sit)e.P-graph
| e.Subgraph-list e.Reduction '$'(e.Sit)e.P-graph

u.Node-context = ('N'e.Node)'X'(e.Context)

e.Interface = '$'(e.Situation) e.P-graph

u.Past-node = s.Index ('N'e.Node) 'G'(e.CP-graph)'E's.Var

u.X-subgraph = 'G'(e.CP-graph)'X's.Var (e.Context-situation)

e.Context-situation = e.Situation

u.Null-subgraph = 'G'(e.CP-graph)'0'

e.Subgr-list = e.Ready-grs 'G'(e.CP-graph)'E's.Var e.Wait-grs

e.Reduction = *empty*
| ('A' e.Assign-list) e.Reduction

e.Ready-grs = *empty*
| u.C-func-assign e.Ready-grs
| u.C-graph-assign e.Ready-grs

e.Wait-grs = *empty*
| 'G'(('N'e.Node)'X'(e.Context))'E's.Var e.Wait-grs

Metacomputation starts with a node-context, where the **e.Node** is the initial node, and **e.Context** is empty. The context in a node-context is referred to as the *internal* context of the node, as distinguished from its *external* context found in the **e.Context-Situation** part of an enclosing X-subgraph. The context **e.Context** in **u.Situation** is also an internal context. The internal context is an integral part of the configuration; it is taken into account when we compare and generalize configuration. The external context does not take part in operations on configurations. The difference between external and internal contexts arises in decompositions, as we shall discuss in a moment. At the beginning, before any decomposition is made, the internal context is empty.

Let the initial node be $< F\ E >$. Then the P-graph for F is read, and the node is transformed into the corresponding situation with empty internal context. We have now an *interface*. This is the format in which the driving takes place, as discussed in Sec. 6.5. If a hindrance $< Q >$ is met in E, a decomposition is made:

$$<FE> = (<Q> \leftarrow e\mathit{V}) / <FE'>$$

This is how an X-subgraph appears:

$$\text{'G'}(\ (\text{'N'}...<Q>)\text{'X'}()\ )\text{'X'}\ \mathit{V}\ (<FE'>)$$

The process of metacomputation enters the subgraph, and the node $<Q>$ is developed in the empty internal context. The external context-situation $<FE'>$ is the situation in the driving of the function F frozen at the moment of decomposition. V is the liaison variable **s.Var**. In the language of Sec. 6.3 we have the stack (q)f.

During the metacomputation of $<Q>$ it may or may not happen that the X-subgraph will be *opened*. If it is opened, the external context $<FE'>$ is added to all situations and nodes inside the subgraph as *internal* context. In the language of Sec. 6.3, on all walk-end in the subgraph we have now the stacks of the form q'f, where q' is any neighborhood that resulted from q. If the subgraph is never opened, its computation is completed independently of the context. We say in this situation that the root configuration of the subgraph is *separated*. The result in the C-graph is a C-assignment to the variable $e\mathit{V}$; it will be **u.C-func-assign** if the whole subgraph is reduced to one function call (as, is the case, e.g., when $<Q>$ is a call of a basic configuration), or **u.graph-assign** otherwise.

Sooner or later, the pointer of the supercompiler will return to the deferred context. If the subgraph was opened, then a passive or partly passive expression may be substituted for the liaison variable $e\mathit{V}$ in the context-situation; otherwise $e\mathit{V}$ in it is treated as a free variable. In any case, we must resume the driving from the moment when it was interrupted by decomposition. The context-situation becomes an interface again. Obviously, we must restore the P-graph exactly as it was before decomposition. In earlier versions of the supercompiler, the current P-graph was saved together with the context-situation. This may require, however, a lot of space, if P-graphs are big and stacks long. Therefore, in the later model of the supercompiler we restore the P-graph from the context-situation and the initial full P-graph for F.

Decomposition and opening create walk-ends of recursive structure which we first introduced in Sec. 6.3. An example: the walk-end ((abc)d)e, where letters stand for some neighborhood has the following structure in the supercompiler (schematically):

$$...\text{'G'}(...\text{'G'}(...\ \mathit{func\text{-}call}(\mathbf{a})\ \mathit{stack}(\mathbf{b,c})\ )\text{'X'}\mathit{sit}(\mathbf{d}))\text{'X'}(\mathit{sit}(\mathbf{e}))$$

The dots represent all other elements of CP-graph except walk-ends. They include contractions (the case dimension), and history records (the time dimension).

Null-subgraphs appear when a Refal step creates a unitary active expression. Before driving it on, we enclose it into the brackets of a null-subgraph. Thus what we denoted as 'a; b; c' in Sec. 6.3 will look as

$$...\ \mathit{hist\text{-}rec}(\text{a})...\text{'G'}(...\ \mathit{hist\text{-}red}(\text{b})...\ \text{'G'}(...\ \mathit{hist\text{-}rec}(\text{c})...)\text{'0'})\text{'0'}$$

The dots stand for contractions, branchings, etc.

The past-node structure is introduced for keeping some nodes in the history records. This is an optional feature. The supercompiler could do without it, using only past situations, and still build a finite graph. But in some cases a past-node makes a short-cut and leads to a shorter program. The node recorded as **u.Past-node** may lie in the mainstream of the development, in which case **s.Var** is **'0'**; or it may be separated as a subgraph, with **s.Var** as the liaison variable.

When a past-node record is left, we do not know whether there will be a looping-back to it. But a number **s.Index** is reserved for the case there will be. As we mentioned, there is no generalization of past-nodes in the supercompiler. If there is a reduction to **e.Node** from one of the descending configurations, it becomes a basic configuration. Otherwise the past-node record is erased when it is passed on the way back and out from the subgraph.

X- and null-subgraphs can only be nested, not parallel. Parallel subgraphs are of the E-type, i.e. have the format:

'G'(e.Subgraph)'E's.Var

We refer to such subgraphs as *separated subgraphs*. They appear in the following four situations.

1. Partly passive top-level walk-end. A simple example of this situation is when we drive the configuration **< Fc e.102 >**, as defined in Sec. 6... (see basic #3), and after one Refal step have the configuration **'D'< Fc e.102 >**. When the walk-end is partly passive, we decompose it, taking out all active subexpressions:

('D' < Fc e.102 > <- e0) =
(< Fc 2.102 > <- e.104) ('D' e.104 <- e0)

The remaining passive expression is formatted as **u.Final-C-assign** and enclosed in parentheses with the flag **'A'**, which is a sign to the driving algorithm to jump over it as a ready component of the final C-graph. The exact computer form is:

```
G((N 105 ()*((Fc )*E 102 ))X())E 104 (A(CD*E 104)E 0)
```

The process enters the subgraph now, and it is soon discovered that the node must be reduced to the root node **< Fc e.102 >**. The final walk-end in the C-graph becomes

```
(F 3 (*E 102 )E 104 (CD*E 104 )E 0
```

If there are more than one subexpressions, the decomposition will create a sequence of parallel subgraphs, e.g.:

('B'(< F1 e1 > (< F2 e2 >'+'< F1 < F3 e1 > >)) <- e0) =
(< F1 e1 > <- e.110) (< F2 e2 > <- e.111) (< F1 < F3 e1 > > <- 112)
('B' (e.110 (e.111 '+' e.112)) <- e0)

The subgraphs will be processed sequentially, thus we have the structure **e.Subgraph-list**, which consists of the ready part of the subgraphs **e.Ready-grs**, the subgraph where the pointer is, and the subgraphs **e.Wait-grs**, which are still to be processed.

It is important that the configuration from which we take out active subexpressions is on the top level of the initial graph or a separated subgraph. If a partly passive expression is found inside at least one pair of activation brackets, we have no need of taking out its active subexpressions: those which are needed for the computation of the enclosing function calls will be found automatically in the process of outside-in driving, and those which are not needed will find their way, uncomputed, into the final answer, i.e. a top-level walk-end, or disappear at some point on this way. When an active subexpression enters this top-level walk-end, there is no choice but to compute it.

2. Reduction to a basic configuration. When we reduce a configuration Q to a basic configuration Q^b, we match the former to the latter and find the reduction assignment (E <- $\underline{var}(Q^b)$). Some of the expressions in the list E may include active subexpressions. As in the preceding case, we have no choice but to compute the values of these before proceeding to the evaluation of Q^b. Indeed, it is in the concept of a basic configuration that its arguments are computed independently of its definition, so that in the metacomputation of the graph of the basic configuration the arguments can be treated as free variables. The overall metacomputation is thus split into two parts. Without this split we could go on infinitely.

Therefore, the active subexpressions from all members of E are taken out and form a list of separated subgraphs. The remaining passive parts are substituted in the arguments of the basic configuration. As in the preceding case, the formula for the CP-walk-end is:

e.CP-walk-end = e.Subgraph-list ('A'u.Final-C-assign)

but while the final C-assignment in that case was passive,

u.Final-C-assign = ('C' e.Passive-expression)'E'0

it is active now:

u.Final-C-assign = ('F'e.Func-name e.Assign-list)'E'0

We have assumed that configuration Q which is reduced to Q^b is simply a specialization of Q^b. We know, however, that this is not the only situation when a reduction to Q^b is necessary. If Q^b, when decomposed into a stack, is ab...c, and Q is a'b'...c'xy...z, where function calls a', b', etc. to c' are specializations of a, b, etc. to c, respectively, we have to reduce Q to a specialization of Q^b in the context xy...z. If we remember that the reduction assignments, again, may include active subexpressions which we must separate as subgraphs, we have the decomposition:

$$(Q \text{ <- } \mathbf{e0}) = \mathbf{e.Subgraph\text{-}list}\ (Q^b_{spec} \text{ <- } eV)\ (Q_{context} \text{ <- } \mathbf{e0})$$

where Q^{b}_{spec} is a specialized call of the basic configuration, and $Q_{context}$ is what remains of Q when the letter it taken out and replaced by the liaison variable eV. In the supercompiler this is represented as

e.Subgraph-list ('A' u.Final-C-assign) '$'(e.Sit)e.P-graph

The ending starting with '$' is a reserved interface corresponding to

$$(Q_{context} \leftarrow e0)$$

As in the case of no context, the action pointer is set at the first separated subgraph; the true interface, therefore, will be there. But sooner or later the pointer will pass the subgraphs and jump over the final C-assignment. Then the reserved interface will become active.

3. Looping back (without generalization). This case does not differ much from the preceding case. When we reduce Q to a past configuration Q^{p}, which can be represented by a situation or by a node, we treat Q^{p} as a basic configuration and reduce Q to it in the same manner. (Q^{p} becomes formally basic when the development of its graph is completed).

Declaring a recurrent configuration basic and the resulting separation of the function calls in the arguments, effectively stops the process of outside-in interpretation of the program at the border-line and replaces it by the inside-out interpretation. Indeed, the separated subgraphs are unconditionally executed first, regardless of whether this is necessary for further computation of the recurrent configuration or not. This change in the order of evaluation affects only the border between the two graphs. Each of the separated subgraphs can include nested function calls, and the evaluation of these calls will start in the usual outside-in order.

4. Generalization. In generalization we reduce Q^{before} to Q^{g}. Again, we take out active subexpressions from the assignments to $\underline{var}(Q^{g})$. Unlike all preceding cases, there will be no final C-assignment; we have the reduction assignments instead (or, rather, what is left of them after taking out active subexpressions). They are parenthesized with the same flag 'A', which tells the driving subprogram to jump over them. At the end we have a reserved interface for driving Q^{g}:

e.CP-walk-end = e.Subgraph-list ('A'e.Assign) '$'(e.Sit)e.P-graph

Let us give an example of the CP-graph at an early stage of work on the problem in Sec. 6.7. The program in the supercompiler which prints out the current state of the CP-graph (for debugging purposes) replaces situations and basic-lists by SIT and BAS, and prints them on separate lines.

```
:((
  1 (N 2 ()*((Fc )*((Fbi )*((Fa )*E 1 ))))
  G(
   G(
    G(
     (BAS)P$(SIT)
BAS= -
SIT= 102 ()((f(Fa ))(*E 1 )E 1 )
```

```
    :((
      S 1 102 $(SIT)
SIT= 103 ()((f(Fa )S 1 2 )(*S 102 )S 2 (*E 1 )E 1 )
      :((
        I 102 A ^^^ (N 103 ()B*((Fa )*E 1 ))X()
       )+(
        (#I 2 A)(=*S 2 *((Fa )*E 1 ))
       ))
     )+(
      X 1 N(=)
     ))
    )X 101 (102 ()((f(Fbi1 )B 1 2 )()E 2 (*HX)E 1 ))
   )X 100 (101 ()((f(Fc ))(*HX)E 1 ))
  )E 0
))
```

The moment we have caught is after the first step of the Refal machine is simulated in driving. The graph starts with a past-node structure fixing the initial node given to the supercompiler. Three nested subgraphs reflect the history of the supercompiler going deeper into the nested structure of activation brackets in search of a call which can be driven without a hindrance. This call is **< Fa e1 >**. The context of this call is remembered as two context-situations of the X-subgraphs: the closer context is made by **Fbi1**, and the farther by **Fc**. A past situation is preceding the development of **< Fa e1 >**, so that if we meet this call again, we will be aware of the recurrence.

The structure of the C-graph already produced reflects the factorization of the P-graph for **Fa**: entering the first branching point, we chop off the first symbol of the argument **e1**, then have a second branching point, which distinguishes between the cases **A** and not-**A**. The second branch of the first branching point take up the case of the empty argument.

The situation in front of the second branching point is a *fork-situation*, not a past-situation. It is not separated from the preceding past-situation by a step, but is a narrowing of it. Indeed, while the neighborhood of the first situation is **f(Fa)**, that of the second is **f(Fa)'S'1 2**, which reflects the path (in the P-graph) which was covered between the two situations.

The three hats ^ ^ ^ represent the pointer. It faces the node we have just obtained by making a step in the function **Fa**: the result is **B < Fa e1 >**. The part of the graph before the pointer is a C-graph, after it a P-graph. One can see the difference in the numbers used as variable indexes. In the C-part, all variables except the input variable **e1** are *new*, and thus have the indexes 101, 102, etc.; the indexes in the P-part are the original indexes 1 and 2.

10. Protection from unwanted generalization

The algorithm of generalization described in Sec. 6.1-4 often results in looping back and generalization which could be avoided without making the graph of states infinite. This algorithm takes into account the recursive structure of the program, and only it. When the initial configuration is full of free variables, as it was in the examples we considered above, our algorithm rightly indicated the moment when we must loop back to avoid an infinity. But when instead of free variables we have specific expressions in the initial configuration, we may (and usually will) wish to postpone looping back, in order to give the supercompiler a chance of partial evaluation of some function calls.

As an example, consider interpretation of a program in a language which features sequential execution of statements and jumps by the GO-TO instruction. We take the simplest case of such a language, and give it the name **L**. A program in *L* is a sequence of statements which are terminated semicolons:

$$S_1; S_2; \ldots S_n;$$

Some statements may be labeled by an identifier which precedes the statement and is separated from it by a colon. A statement is either a *regular* statement which makes some transformation of the current state of the computing system and passes control to the next statement, or by a statement of the form

'COND-JUMP 's.Label

which either does nothing, or sends control to the statement labeled by **s.Label** (conditional jump). We do not care here about the syntax of regular statements; we only assume that they do not include colons or semicolons on the top level of parenthesis structure.

To define the semantics of **L**, we define in Refal a recursive function which interprets a given program in **L** with a given set of input data. Let this function's name be **L** too. An important part of the semantics of such languages as **L** is the concept of a *state* of the computing system, which changes as computation proceeds. The state will usually include the values of all variables defined at the moment. The details, again, are not important for our purpose. The input data can be treated as a part of the initial state. Then the function **L** must depend on the state and the program, and show how the program causes the state to change.

Besides the values of variables, there is one more element of the state of the system: the position of the *control point* which shows the next statement to execute. Thus **L** will have the format:

< L (e.Past-prog) e.Prog (e.State) >

Here **e.Past-prog** is the part of the program which has been passed already, but we still have to keep it because a GO-TO statement can take us to any point in the program. The initial call of **L** is

< L () e.Program (e.Initial-state) >

Here is its definition in strict Refal:

```
* The interpreting function of the language L
L {
*1. Jump over a label
   (e1) s.Lab':'e2 (e.State)  =  < L (e1 s.Lab':'e2 (e.State) >;
*2. Execute a C-JUMP statement. First check the condition.
   (e1)'C-JUMP 's.Lab';'e2 (e.State)  =
         < C-jump (e1)'C-JUMP 's.Lab';'e2( < Cond e.State > ) >;
*3. End of program. Print out the state as the output.
   (e1) (e.State)  =  < Prout e.State >;
*4. Scan and execute a regular statement.
```

```
    (e1) e2 (e.State)  =  <Scan (e1)()e2 (e.State)>;
  }

* Scan a regular statement for its end, and execute.
Scan {
*1. Semicolon found. Execute the statement.
    (e1)(e.S)';'e2 (e.State)  =
            <L (e1 e.S';') e2 (<Exec-s e.S (e.State)>)>;
*2. Recursion. A symbol.
    (e1)(e.S) sX e2(e.State)  =  <Scan (e1)(e.S sX) e2 (e.State)>;
*3. Recursion. Parentheses.
    (e1)(e.S) (eX)e2 (e.State)  =  <Scan (e1)(e.S(eX)) e2 (e.State)>;
*4. Error in the program. Print a message.
    (e1)(e.S) (e.State)  =
            <Prout 'Error. No semicolon after statement' e.S>;
      }

* Conditional jump to a label.
C-jump {
*1. Condition is false. Go on.
    (e1)'C-JUMP 's.Lab';'e2 (0 e.State)  =
            <L (e1'C-JUMP 's.Lab';')e2 (e.State)>;
*2. Condition is true. Send to Jump.
    (e1)'C-JUMP 's.Lab';'e2 (1 e.State)  =
            <Jump s.Lab ()e1'C-JUMP 's.Lab';'e2 (e.State)>;
        }

* Jump to a label.
Jump {
*1. The label is found
    s.Lab (e1) s.Lab':'e2 (e.State)  =  <L (e1 s.Lab':') e2 (e.State)>;
*2. Recursion. A symbol.
   s.Lab (e1) sX e2 (e.State)  =  <Jump s.Lab (e1 sX) e2 (e.State)>;
*3. Recursion. Parentheses.
   s.Lab (e1) (eX) e2 (e.State)  =  <Jump s.Lab (e1(eX)) e2 (e.State)>;
*4. Error. No label.
    s.Lab (e1) (e.State)  =
            <Prout 'Error. No label ' s.Lab>;
      }
```

The function **Cond** examines the state (e.g. the value of a certain variable, say I) and puts **1** in front of it if the condition is true, or **0** otherwise. The function **Exec-s** executes one statement in a given state. Its value is the state resulting from the execution of the statement. The exact definition of these two functions does not matter.

Let us give to **L** a completely defined program, e.g.:

```
A=B+C; Lab1: I=I-3; C-JUMP Lab1; I=I-1; C-JUMP Lab1;
```

where we just put some plausibly looking strings for statements. The input data is unspecified, and so is, of course, the state of the computing system. We want to meta-evaluate the call of L under these circumstances, in order to compile the program (see Sec. 1.8). The initial configuration is

(Q_1) **< L () A = B + C; Lab1: I = I-3; C-JUMP Lab1; I = I-1; C-JUMP Lab1; (e1) >**

Let us now trace the work of the supercompiler. Configuration Q_1 is *transitive*: it goes over into the next configuration:

< Scan ()() A = B + C; Lab1: I = I-3; C-JUMP Lab1; I = I-1; C-JUMP Lab1; (e1) >

without any contractions. Therefore, the supercompiler will leave no past situation for Q_1, and there will be no attempt to loop back to it. In the same manner, a number of following configurations, which appear as **Scan** looks through its completely defined argument, will be transitive and leave no history records. The supercompiler works as a partial evaluator.

After finding the end of the first statement, we come to the configuration:

< L (A = B + C;) Lab1: I = I-3; C-JUMP Lab1; I = I-1; C-JUMP Lab1; (< Exec-s A = B + C; (e1) >) >

According to the semantics of L, which is defined in the standard Refal with the inside-out evaluation order, the statement A = B + C must be now executed over the unknown state **e1**. However, our supercompiler is an outside-in evaluator. At this moment it does not need the value of **Exec-s**. It will go on with scanning the program, and will do this until some of the statements in the program -- it will be, of course the conditional statement C-JUMP -- requires the value of the state for doing the next Refal step.

As we discussed before, the switch to outside-in evaluation often leads to an improved program. On the other hand, we often want to stick to the exact semantics of the language we deal with, L. This can be achieved by using certain tricks in the context of the general outside-in supercompiler, which we will consider in Chapter 7; or we can use a version of the supercompiler which sticks to the inside-out evaluation order. In the following we go on with the standard -- for the supercompiler -- outside-in order.

As the reader can verify, a sequence of transitive configurations will take the process to the configuration:

< C-jump (A = B + C; Lab1: I = I-3;)
C-JUMP Lab1; I = I-1; C-JUMP Lab1;
(< Cond < Exec-s I = I-3 (< Exec-s A = B + C(e1) >) > >) >

Now the calls of **Cond**, and then, sequentially, the two calls of of **Exec-s** become hindrances, which causes the decomposition:

(< Exec-s A = B + C (e1) > <- e.103)
(< Exec-s I = I-3 (e.103) > <- e.102)
(< Cond e.102 > <- e.101) Q_2

(Q_2) **< C-jump (A = B + C; Lab1: I = I-3;)**
C-JUMP Lab1; I = I-1; C-JUMP Lab1; (e.101) >

Next steps depend on the definition of **Exec-s** and **Cond**. Since we have left these functions undefined, the supercompiler will not find their definitions in the program and automatically declare them basic, leaving their calls in the final C-program. Then it will proceed with the development of Q_2.

Note that as the result of partial evaluation and decomposition, the program has taken on a typical structure of program written in a command language: execute the first statement and assign the value to **e.103**; execute the second statement (using **e.103**) and assign to **e.102**, etc. It can be easily translated into a machine-oriented language. The scanning of the program has been made, and will not be repeated at the execution time.

Unlike all preceding configurations, Q_2 is not transitive. It is kept as a history record. When Q_2 is further developed, two branches appear. On the first branch (false condition, **e.101 -> 0 e.101**) we proceed with scanning the program and come to the next configuration of the same kind:

(Q_3) **< C-jump (A = B + C; Lab1: I = I-3; C-JUMP Lab1; I = I-1;)**
C-JUMP Lab1; (e.105) >

It is here that the unwanted generalization will take place, if we do not prevent it. We know that there must be no looping-back here; we should go further scanning the program until we register all sequential statements and identify all loops. But Q_3 belongs to the same neighborhood as Q_2. The algorithm of generalization, as we defined it above, will demand looping back. The generalization of Q_3 and Q_2 will result in the configuration

(Q^9) **< C-jump e.106 (e.105) >**

which will be developed instead of Q_2. All information about the program is lost in Q^9. The metacomputed program from this point will simply interpret the program, repeating the definition of **L**.

The solution of this problem in the supercompiler is a special interpretation of *delayed metacoding* (see Sec. 3.9). All graphs and their elements are downgraded to metacode when they are handled by the supercompiler. When we form the initial configuration Q_1, we do not actually downgrade the program, but use delayed metacoding. The metacode of Q_1 becomes:

***(L () *!(A = B + C; Lab1: I = I-3; C-JUMP Lab1; I = I-1; C-JUMP Lab1;)**
(*E 1))

All subprograms of the supercompiler recognize that ***!**(E) stands for the metacode of the object expression E, and behave correspondingly. Besides that, delayed metacoding serves as protection from generalization. If an operation of generalization requires a replacement of two unequal protected expressions by a free variable, this operation is not permitted. A looping-back which requires such a generalization will not be executed, and the driving will go on instead. A generalization of two identical protected expressions is, of

course permitted, and the result will be the same expression -- protected, again. If an expression is protected, all its subexpressions, possibly separated and handled by different functions, will also be protected.

In our case, the generalization of Q_3 and Q_2 is not permitted, because it requires to generalize

```
A=B+C; Lab1: I=I-3; C-JUMP Lab1; I=I-1;
```

with

```
A=B+C; Lab1: I=I-3;
```

Therefore, the driving will go on, as it should. On the other hand, when the supercompiler explores the second branch which starts at Q_2, **(e.101 -> 1 e.101)**, the configuration Q_2 returns exactly as it was before, except for the names of variables. There will be no obstacles for looping back, and the metacomputation will be finite.

The general rule of using generalization protection is: protect those object expressions which you want to be incorporated into the structure of the future program, and not interpreted. Generalization protection is a way of controlling the process of metacomputation. The dark side of this method is that there is no more guarantee that the process will terminate. If protection of some expression results in infinite metacomputation, or too big a program, this protection can be removed.

6.11 Freezer

Consider a configuration Q which has no free variables. It is, certainly, transitive and, therefore, leaves behind no history record. All the supercompiler is doing is a trivial step of driving over Q, simulating one step of the Refal machine. This is, however, a costly simulation. It will typically take a hundred of Refal steps -- and only in order to make one Refal step. If we can revert in such cases to a direct execution of Refal step over Q, it could mean a significant gain in efficiency.

One way to do it is this: before every step of driving examine the configuration, and if it has no free variables (ground configuration), upgrade it from metacode into the main code, and pass to the Refal interpreter. Since a ground configuration remains ground after any number of steps, the Refal interpreter should be allowed to work in the usual manner until the result is a passive expression. Then the result must be downgraded and passed back to the driving function of the supercompiler.

This can be easily done, but we can do more. The cases when the whole configuration includes no variables are, usually, of a secondary importance; this happens only to some parts of the overall configuration, because otherwise there would be no need to call the supercompiler. But it happens very often that even though there are free variables in the configuration, they have no role to play along considerable stretches of computation, so that we have a sequence of transitive configurations, as if there were no variables. We have seen an example of this situation in the preceding section, where the configuration Q_1:

< L () A = B + C; Lab1: I = I-3; C-JUMP Lab1; I = I-1; C-JUMP Lab1; (e1) >

includes a free variable, but still gives rise to a row of transient configurations as the program is scanned in a search for a semicolon. The driving procedure does not even notice the existence of the variable **e1**. The same situation will take place later when **Exec-s** makes a parse of a statement.

We want to take advantage of such situations in order to conduct metacomputation more efficiently. We want a system which automatically adjusts the metasystem level of computation: wherever metacomputation becomes a simulation of computation, the system must switch to the lower level and do direct computation. When direct computation becomes impossible because of variables, the system must call the function which does metacomputation.

We achieve these automatic switches by using the idea of a *freezer*. We endow the implementation of Refal with the following additional capabilities.

1. An additional type of object is allowed in the view-field, which will be referred to as a *object variables*. The information content of an object variable is its type (**s**, **t** or **e**) and its index (a symbol). As long as the values of object variables are not required for execution of a Refal step, they are passed from one expression to another and may be copied like object expressions; their existence, or rather their difference from object expressions, is not noticed by the system. When the step becomes impossible because of an unknown value of an object variable, a *freeze* takes place.
2. The function call in the view-field which caused the freeze must be located within the argument of the special built-in function **Freezer**:

< Freezer ... **< F ...** *trouble-spot* **... > ... >**

Freeze is then executed as follows. First the enclosing call of **Freezer** is found. If there are more than one, the innermost call is taken; if there are none, an error condition occurs. Then the whole argument of **Freezer** is downgraded to metacode, and control is passed to this call of **Freezer**.
3. Functions **Dn** and **Up** are extended to include operations on object variables:

< Dn obj-var**(s.Type, s.Ind) >** **=** **'*'s.Type s.Ind**
< Up '*'s.Type s.Ind > **=** obj-var**(s.Type s.Ind)**

Note that even though object variables stand essentially for the same things as free variables, we had to introduce a special notation for them in order to express the operations above as Refal sentences. Indeed, something like

< Up '*EX' > **=** **eX**

would violate the syntax of Refal (a free variable in the right side which is not found in the left side), while

< Dn eX > **=** **'*EX'**

would define **Dn** as a completely different function.

obj-var('E','X') and **eX** are, essentially, identical. It is only in Refal sentences that they are not interchangeable, because of a special use of free variables in Refal. Object variables can be compared to imaginary numbers in algebra. When we limit ourselves to functions defined on object expressions only, as all Refal functions are, function **Dn** is defined everywhere, like squaring a real number, but the reverse function **Up** is not defined on such arguments as **'*EX'**, like the extraction of a square root is undefined on negative numbers. We extend the definition of **Up** by introducing "imaginary" object expressions.

3. Function **Freezer** is defined as follows. If it gets control as a result of a freeze then it puts **1** in front of the argument, i.e. for this case

< Freezer eX > = 1 eX

If it gets control as always in Refal, because the argument is computed and is passive, then

< Freezer eX > = 0 eX

The supercompiler uses the "freezer" **Freezer** in the following way. Whenever a new node is formed in a step, the function **Devlp**, which is to develop a graph for it, upgrades the configuration to the main code, puts it in the freezer, and calls function **Chfrzr** which checks the outcome of the freezer:

```
Devlp {...
   [ e1 ^ ('N'sN(eR)'*'((sF) eE))'X'(eX) ]  =
         <Chfrzr [ e1 ^ ('N'sN(eR) <Freezer <Up eE>> )'X'(eX) ]>;
...
```

Note that when **eE** is upgraded (demetacoded), its big subexpressions, like the program in **L** in the example above, are inside delayed-metacoding brackets; they are demetacoded in one step, without scanning the expression.

The result of the upgrading of **eE** is active, so its direct evaluation is started by the Refal interpreter. For evaluation to become possible, the program which is loaded into the Refal machine must include not only the functions of the supercompiler, but also the "object" functions of the graph G^P (the latter are not necessary if the freezer is not used).

Function **Chfrzr** is define as follows (with small simplifications):

```
* Check freezer
Chfrzr {
*1. Lower-level computation completed. Freezer contains
*   a passive expression in the main code.
  [ e1 ^ ('N'sN(eR) 0 eE )'X'(eX) ] =
      <Step [ e1 ^ ('N'sN(eR) <Dn eE> )'X'(eX) ];

*2. Lower-level computation aborted. Freezer contains a metacoded
*   configuration
  [ e1 ^ ('N'sN(eR) 1 eE )'X'(eX) ] =
         <Step  [ e1 ^ ('N'sN(eR) eE )'X'(eX) ];
      }
```

As the the direct evaluation goes on, one of the two outcomes will take place (not counting infinite loops in the program), which is reflected in the two sentences for **Chfrzr**. The evaluation may go through completely. Then the result is metacoded, and a new node is formed and passed to the function **Step** which decides what to do next. Or the evaluation may abort on some stage. Then it is continued on the level of metacomputation.