Chapter 3

THE BASICS OF METACOMPUTATION

When we use the Refal machine for computation, we load a program into it, put the expression to evaluate into the view-field, and switch the machine on. The Refal machine works on its own; we do not interfere with its operation. After a while it comes up (or does not come up) with the desired (or not exactly desired) result.

Metacomputation is an operation mode of the Refal machine where it is not left to itself, but is run under the supervision and control of a higher authority, which can be the user, i.e. a human being, or another machine. The purpose of this is to explore the behavior of the Refal machine with an initial view-field which is not completely defined, but can be represented by an expression with free variables. The program is definite and remaines fixed. The ultimate goal of metacomputation is to construct a formal model of the Refal machine working under these circumstances, a model which could serve as the machine's alternative definition for that special case.

3.1 Examples of driving

Let us start with a couple of simple examples which will help us in defining the basic concepts of metacomputation in Refal.

Consider the definition of the power function through repeated multiplication:

```
Power {
  1 Of sX  =  sX;
  sN Of sX  =  <Mul <Power <Sub sN, 1> Of sX>, sX>; }
```

Suppose now that we want to compute the cube of some number **sX** using the function **Power**:

(Q_1) **<Power 3 Of sX>**

We can do it directly by turning the expression above to the Refal machine. But we can do better. Since the variable **sN** of the function **Power** has a definite value, we can do some part of computation in advance, namely that part which depends on **sN**, but does not depend on **sX** (which remains unknown). This procedure is known as *partial evaluation* of the function call Q_1. It is the simplest form of meta-evaluation.

Indeed, let us put the definition of **Power** in the program field of the Refal machine, and Q_1 in the view-field. We must note immediately that a non-ground (i.e. including free variables) expression like Q_1 can not be actually placed in the view-field. This only is our way of tracing the evolution of

a whole class of ground expressions in the view-field of the Refal machine. We shall refer to classes of ground expressions in the view-field as *configurations* of the Refal machine. Thus the expression Q_1 is a configuration. It represents the set of ground expressions the members of which are all expressions that can be obtained from Q_1 by substituting some symbol for **sX**. The program in the the program field of the Refal machine is fixed and does not change when the machine is running. Therefore, the ground expression in the view-field determines, at every moment, the exact state of the Refal machine. A configuration defines a *generalized* state of the Refal machine, a set of exact states. Since the Refal machine is deterministic, its exact state determines the exact state after the next step, and all the subsequent steps. For a generalized state represented by a non-ground expression, the next state may depend on the values of the variables in the expression. Thus the evolution of the view-field must be represented by a graph, where the nodes are configurations, and the directed arcs describe transitions between the configurations and carry conditions on the configuration variables. We shall refer to this graph as *the graph of states* of the Refal machine, or simply a Refal graph. The result of metacomputation should be a Refal graph.

So, let us construct the graph of states for the initial configuration Q_1. What will be the next state after Q_1? To find it, we must make a step of the Refal machine with a non-ground expression in the view-field. We call such a step *driving*. The free variables of the non-ground expression are not supposed to get into the cogwheels of the Refal machinery, but we still "drive them through" forcefully, treating them as sets of their admissible values.

In accordance with the general step procedure, we match the argument of **Power** to the left side of the first sentence:

3 Of sX : 1 Of sX

Such a matching is something new for us, because the argument (left side) of it includes a free variable **sX**. Note that **sX** in the argument has nothing to do with **sX** in the pattern. Their textual identity is purely coincidental. The left-side **sX** represents the set of all symbols which can replace it in the current configuration; the right-side **sX** is a detail of the Refal machine used for attaching to it a value, which can be any symbol. We are going to handle matching and replacement in such a way that the use of textually identical variables in these two sets will never lead to a confusion.

The result of matching in our case is pretty obvious: it never (i.e. for no value of **sX**) succeeds, because **3** does not match **1**.

Thus the Refal machine will try to use the second sentence, i.e. to match:

3 Of sX : sN Of sX

Here the matching is as easy: it always (i.e. for any value of **sX**) succeeds, and **sN** takes the value **3**, while **sX** takes the value **sX**. Therefore, the Refal machine makes a replacement and the next configuration in the view-field becomes

(Q_2) **< Mul < Power < Sub 3, 1 > Of sX >, sX >**

In the next step, **< Sub 3, 1 >** becomes primary active, and is replaced by **2**:

(Q_3) **< Mul < Power 2 Of sX >, sX >**

The next step is analogous to the first one, and the result is:

(Q_4) **< Mul < Mul < Power < Sub 2, 1 > Of sX >, sX >, sX >**

Proceeding in this manner, we have:

(Q_5) **< Mul < Mul < Power 1 Of sX >, sX >, sX >**
(Q_6) **< Mul < Mul sX, sX >, sX >**

And this is all we can do. Further metacomputation is impossible, because multiplication is a built-in function which has no Refal definition. The calls of this function will have to enter our final Refal graph. The configurations which, for one reason or another, are allowed to enter the final graph of states will be referred to as *basic*. One reason for a function call to become a basic configuration is that the function is built-in.

So, we have completed the partial evaluation and come to the expression Q_6 for the cube of sX. As a program, it is considerably more efficient than the initial program Q_1. There is no need now to make comparisons and subtractions; we only have to perform two multiplications.

In our second example there are no predefined parameters which would make a case for partial evaluation. An optimization of a program will be achieved by metacomputation through a change in the control structure. Consider this program:

```
Fab { A e1 = B < Fab e1 >;
      s2 e1 = s2 < Fab e1 >;
      = ; }

Fbc { B e1 = C < Fbc e1 >;
      s2 e1 = s2 < Fbc e1 >;
      = ; }

F { e1 = < Fbc < Fab e1 > >; }
```

The function **F** first applies **Fab** to its argument, which substitutes **B** for every **A**; then it applies **Fbc**, which substitutes **C** for every **B**. Let us meta-evaluate the call of **F** with an arbitrary argument:

(Q_1) **< F eX >**

As the result of the first step we have

(Q_2) **< Fbc < Fab eX > >**

At the next step **< Fab eX >** becomes primary active, and we have to match:

eX : A e1

In the next section we shall give the general algorithm for matching non-ground expressions. In this section we perform matching by reasoning for each specific case. The variable **eX** in the argument of the matching may have any value. If it starts with the symbol **A**, the matching will be successful, otherwise it will fail. Tracing the evolution of the state of the Refal machine, we must consider these two cases separately. If the the value of **eX** is such that the matching succeeds, the first sentence in the definition of **Fab** will be used. For other values of **eX** we must examine the applicability of further sentences. Therefore, at this stage of tracing we should separate from the whole set of expressions a subset for which the matching is successful, and the first sentence applicable. We *narrow*, or *contract* the whole class represented by a non-ground expression (**eX** in this case) to a subclass which includes those and only those ground expressions for which the first sentence of **Fab** will be used. We call this operation a *contraction*.

Contractions make one of the two types of building blocks from which Refal graphs are constructed. A digression to describe these blocks is now in order.

The general form of a contraction is

v -> *E*

where *v* is a variable, and *E* a pattern expression which represents the subset to which the set of all admissible values of *v* is narrowed. In our case, the contraction can be written as

eX -> A eY

In addition to checking a certain condition on the value of a variable, a contraction matches that value to the pattern in its right side, thus assigning some values to the variables of the pattern. We say, therefore, that in the contraction *v* -> *E* the variables of *E* are *derived* from (or are *derivatives* of) the variable *v*. Example: if the value of **eX** is **A B C D**, then the contraction above is applicable and results in the value **B C D** for the variable **eY**. We can use the contracted variable itself in the right side of the contraction, e.g.,

eX -> A eX

The execution of this operation, when it is successful, results in the reassignment of a new value to **eX**.

A contraction is, actually, a special case of the general matching operation when the left side is a single variable. We write it with an arrow because with a single variable in the left side a matching becomes a substitution: that of a pattern for a variable. The other special case of matching is with a single variable in the right side. This substitution is an *assignment*:

E <- *v*

An assignment is always applicable if the syntactic type of *E* corresponds to the type of *v*. We shall use only such assignments where this condition is satisfied. The assignments like

A1 + e1 <- tX

which are always failing, or

e1 <- s5

which may fail or succeed, will be considered illegitimate.

Our notation of general matchings, contractions, and assignments keeps to the following two principles:

1. The variables on the left (argument) side are *defined*, i.e. have some, known or unknown, values. The variables on the right (pattern) side get defined in the process of matching through the values of the left-side variables.
2. When one of the two parts to the matching is a single variable, the matching becomes a substitution, and the arrow is directed from that variable to its replacement. Assignments will also be used as a form to list the values of defined variables.

Here are a few examples of the execution of contractions and assignments.

The values of defined variables	Operation	The values after the operation
A B C D <- eX	**eX -> A eX**	**B C D <- eX**
(A B C) <- e2	**e2 -> (e1 sX)e2**	**A B <- e1** **C <- sX** **<- e2**
(A B (C)) <- e2	**e2 -> (e1 sX)e2**	*failure*
'+*+' <- e1**	**e1 -> sX e1 sX**	**'+' <- sX** **'***' <- e1**
A B <- e1 **C D <- e2**	**e1 e2 <- e1**	**A B C D <- e1**

We can return now to the driving. The configuration we consider is Q_2, and we separated a subclass of it which is described by the contraction **eX -> A eX** and includes all those ground expressions of Q_2 for which the Refal machine uses the first sentence of **Fab**. Now we do the same for the second and the third sentences, which call for the contractions **eX -> sY eX** and **eX -> ** , respectively. Therefore, there will be three branches outcoming from the configuration Q_2 in the Refal graph, as shown in Fig. 3.1. The symbolic form we shall use to represent Refal graphs is close to the form of a Refal program. The generalized history of computation as we have traced it by this moment, and as pictured in Fig. 3.1, is represented as follows:

$Q_1 = Q_2$ { **(eX -> A eX)** $= Q_3$
+ (eX -> sY eX) $= Q_4$
+ (eX ->) $= Q_5$
}

Here the equality sign symbolyzes, as in a Refal program, the replacement of one expression by another. The configuration Q_1 turns into Q_2 unconditionally. Q_2, however, finds itself at a branching point. The braces enclose the list of branches starting from the point represented by the opening brace. The list is written as a sum. A branch includes a number of parenthesized contractions and is concluded by a replacement. Like the sentences of a Refal program, the branches starting at each node are ordered. A branch is actually used only if all preceding branches were found unapplicable. The subclass **sY eX** of the second branch of the graph includes the subclass **A eX** of the first branch. But when we interpret this graph as a program, this contraction will be used only if **sY** is not **A**.

The configurations Q_3, Q_4, and Q_5, are as follows:

(Q_3) **< Fbc B < Fab eX > >**
(Q_4) **< Fbc sY < Fab eX > >**
(Q_5) **< Fbc >**

We now have to trace the further development of each of these configurations. In the case of Q_3, the primary active expression in the next step will be **< Fab eX >** again. However, we can force the Refal machine to work differently. Consider the outer function **Fbc** in Q_3. It will not be evaluated until the computation of **Fab** is completed. But we can figure out that the first step in the evaluation of **Fbc** does not depend on the result of the computation of **< Fab eX >**. We can change the order of evaluation of function calls without changing the overall result. Let us consider **< Fab eX >** as some unknown expression and make one step in the evaluation of Q_3. The result will be

Q_3 **= C < Fbc < Fab eX > >**

We can do the same with Q_4. The result of the step will depend on **sY**, so there will be a branching on its value, but there is no dependence on the value of **< Fab eX >**:

Q_4 **{ (sY -> B) = C < Fbc < Fab eX > >**
+ = sY < Fbc < Fab eX > >
}

As for Q_5, it becomes empty. Combining all this in one graph yields:

Q_1 **= Q_2 {(eX -> A eX) = Q_3 = C < Fbc < Fab eX > >**
+ (eX -> sY eX) = Q_4 { (sY -> B) = C < Fbc < Fab eX > >
+ = sY < Fbc < Fab eX > >
}
+ (eX ->) = Q_5 = *empty*
}

We could drive further each of the active configurations in this graph, and go like that infinitely. We want, however, a finite model of our system. We can notice that on the ends of all branches we have either a passive (actually,

empty) expression, or the combination **< Fbc < Fab eX > >**, which is nothing else but the configuration Q_2! We can, therefore, transform our graph as follows:

```
Q1 = Q2 { (eX -> A eX) = C Q2
        + (eX -> sY eX) { (sY -> B)  = C Q2
                        + = sY Q2
                        }
        + (eX -> ) = empty
        }
```

This graph has two configurations, Q_1 and Q_2, as its nodes. For each of them the graph includes a *development*, i.e. a subgraph indicating what happens to the configuration after one or more steps of the Refal machine. Therefore, this graph is a self-sufficient model of the Refal machine evaluating the initial configuration Q_1. It can be used to compute Q_1, i.e. **< F eX >**, with any value of **eX**; we can convert this graph into an equivalent Refal program.

Of the two configurations in this graph, Q_1 goes over, unconditionally, into Q_2. We shall call such configurations as Q_1 *transient*. We do not really need Q_1 in the graph; it can be identified with Q_2. The other configuration, Q_2, must necessarily stay in the graph, because it calls itself. Q_2 is a *basic* configuration. Calling itself is a good reason for a configuration to be declared basic and stay in the final graph.

Identifying Q_1 with Q_2, and rewriting our Refal graph in the format of a Refal program, we get a new definition of the function **F**:

```
F { A eX = C <F eX>;
    B eX = C <F eX>;
    sY eX = sY <F eX>;
    = ; }
```

This is a better definition than the original one. With the original program the argument was passed twice; now all work is done in one pass of the argument; instead of converting every **A** into **B**, and then **B** into **C**, **A** is converted directly into **C**.

3.2 Strict Refal

Metacomputation is considerably simplified if we put certain restrictions on the language in which the program is written. We shall refer to the language thus restricted as *strict* Refal. It is a subset of basic Refal. The restrictions are:

1. t-variables must not be used;
2. only such built-in functions are allowed that the effects of their evaluation, including the values returned, do not depend on the order of evaluation;
3. in the left sides of sentences there must be no open e-variables and no repeated e-variables;

The first requirement does not ask for much, because wherever a t-variable, say **tX**, is used, we can eliminate it by considering separately the two possible cases of **tX**: when it can be replaced by **sX**, and by **eX**. For instance, the sentence:

< F (e1) tX e2 > = < F (e1 tX) e2 >

can be replaced by two sentences:

< F (e1) sX e2 > = < F (e1 sX) e2 >;
< F (e1) (eX) e2 > = < F (e1 (eX)) e2 >;

However, if there are *n* t-variables in a sentence, the replacement will produce 2^n sentences, and this may be a nuisance. Therefore, it is advisable to avoid t-variables when writing with a view on metacomputation. This is not very hard, because the programmer using a t-variable usually knows whether a symbol or a parenthesized expression is expected. If he does not know this, he should modify his data structures so as to know, or use no more than one t-variable in a sentence.

The restriction on built-in functions is called for by the outside-in order of evaluation in metacomputation. We must be able to change the order of evaluation of function calls without altering the results of computation. In particular, Input/Output functions, as well as the functions **Br** and **Dg**, cannot be used. Suppose, e.g., that we meta-evaluate the following expression:

(1) **< F1 (< Input 1 >) < Input 1 > >**

where the function **F1** is defined as follows:

F1 { (e1) 0 e2 = e1;
(e1) e2 = e1 e2; }

Driving (1) outside-in, we match:

(< Input 1 >) < Input 1 > : (e1) 0 e2

e1 takes on the uncomputed value of the first call **< Input 1 >**, but further matching requires the evaluation of the second call **< Input 1 >**, in order to test whether it starts with **0**. Therefore, the first function call to be evaluated is the second call of **Input**, while it is the first call that must be evaluated first with the inside-out order. The program resulting from meta-evaluation will not be equivalent to the original program, because the first item in the input stream of channel 1, which is meant by the original program for the first argument of **F1**, will be directed to the second argument.

However, if we meta-evaluate

(2) **< F1 (< Input 1 >) < Input 2 > >**

there will be no such confusion, and the metacomputed program will be equivalent to the original one. If there are no more inputs than one from each channel, the independence of the results from the evaluation order required by Restriction 2, is guaranteed. The same is true with regard to the functions

Dg and **Cp** if the names under which expressions are buried are fixed. Calls of **Cp** can be evaluated any number of times. In this way we can keep some global paramenters buried, while still writing in strict Refal.

Beyond these simple cases, Input/Output and Bury/Dig functions cannot be used in strict Refal. They must be left as they are in the program, and only such functions picked up for transformation by metacomputation that can be defined in strict Refal.

The difference between the inside-out and outside-in orders of evaluation is the question of semantics. We chose to endow our basic language, Refal, with the inside-out semantics, but execute metacomputations over Refal expressions using the outside-in evaluation order. It is because of this semantic difference that we have to introduce restrictions on built-in functions. Our choice may seem strange, so we want to give some arguments in its favor.

Our main argument has already been mentioned above: the inside-out semantics is much easier to understand. The order in which function calls are evaluated in a given expression with free variables is uniquely (and simply) determined in the case of the inside-out semantics. With the outside-in semantics, the order of evaluation may depend on the values of free variables, and there are no universal algorithms which could answer simplest questions about actual evaluation order, e.g., whether for any values of free variables one function call will be evaluated before another. We consider *the symbolic process* as our most fundamental primary concept; we want, therefore, our notation to represent processes in an easily understandable way, and we must be able to combine processes in natural ways.

A user who intends his program for metacomputation can write in Refal with the outside-in semantics. Then he can use any built-in functions. But it will be much more difficult than in the case of the inside-out semantics. In the example above, e.g., he would have to figure out that the second call of **Input** is called first, and put the data into the input stream in the corresponding order. This assumes, however, that the call (1) is on the top of the graph of states. What if it is not on the top? It may me called after some other input operations, or not called at all, if its value is not actually needed. To understand the meaning of what he is writing, the programmer may have to do a lot of analysis. When we program in an operator language, we cannot but think in terms of the order of operations. When we use a functional language, we think in terms of pairs argument-value, and want to minimize the interaction between computation processes. The semantics of Refal is the simplest procedural interpretation of a functional language in this spirit.

There is another argument in favor of the inside-out semantics, which is of no less importance. We use the outside-in order in metacomputation for the sake of efficiency. But it is not always true that an outside-in execution of a program is more efficient than the inside-out. Consider this example. Evaluate

(1) **< F1 < F2 e1 > >**

where the definition of **F2** is not important, and **F1** is defined by

F1 { eX = eX eX; }

With the inside-out order, we evaluate **< F e1 >** and then copy the result and concatenate it to itself. With the outside-in order, we copy *the function call* and concatenate it to itself:

< F2 e1 > < F2 e1 >

and then compute **< F2 e1 >** twice. Obviously, this is not the efficient way.

Therefore, in metacomputation we should be free at any moment to switch to whichever of the two methods leads to better results. The meta-evaluator described in Chapter 6 (the supercompiler), when developing the graph of states for (1), will depart from its regular outside-in order, so as not to compute the same function call twice.

Thus our solution to the problem of evaluation order is this: we take the inside-out order as the semantics of Refal, and consider any deviation from it as a way of optimization.

We shall still have a few occasions to return to the evaluation order. We shall discuss in more rigorous terms the relation between the two semantics, and how the outside-in semantics can be interpreted in terms of the inside-out Refal machine operating on infinite objects (Sec. 3.5). And we shall see how an outside-in meta-evaluator can be forced to follow the inside-out semantics (Chapter 7).

We turn now to the third restriction imposed on the strict Refal. Its meaning is to make all recursion explicit by eliminating hidden, implicit recursion that may be present in the process of pattern matching.

Indeed, in the program

```
One-a-b {
   e1 A e2  =  e1 B e2;
   e1  =  e1; }
```

it may seem that **One-a-b** is a non-recursive function. Actually, of course, it includes a loop, which is implicit in the open e-variable **e1**. In the same manner, there is a hidden recursion in the first sentence of the function

```
Equal {
   e1,e1  =  T;
   e1,e2  =  F; }
```

We shall use the term *L-expression* (from *L*eft side of a sentence) for a pattern expression which has no open or repeated e-variables. An L-expression has no more than one e-variable on every level of parentheses structure, and each e-variable may be used only once throughout all levels. The left sides of sentences in a strict Refal program must all be L-expressions. The following property of L-expressions should be noted: any subexpression of an L-expression is an L-expression. But, if an *L-class* is a class represented by an L-expression, then a subclass of an L-class is not necessarily an L-class. While a subexpression is a *part* of the original expression, a subclass is represented by an expression which can be obtained from the original expression by a *substitution*. Thus **e1** is an L-class, but its subclass **e1 + e2**, which is obtained by the subsitution **e1 -> e1 + e2** is not.

There is a Refal program which translates a text in Refal, including extensions, into strict Refal. We have had already some experience with eliminating hidden recursion (see Sec. 2.2, and exercises), and saw that it can be easily algorithmized. In the case of **One-a-b**, we want to eliminate the open variable **e1** in the first sentence. We do that by putting it in parentheses, which means that it will be a different function. The left side becomes:

(1) **< One-a-b-1 (e1) A e2 > = e1 B e2**

The function **One-a-b** must call this new function with the initial value of **e1** empty:

< One-a-b eX > = < One-a-b-1 () eX >

Each open variable in a sentence calls for a special funciton which defines the process of its lengthening. One sentence, (1) in our case, mimicks the original sentence; two others describe lengthening:

(2) **< One-a-b-1 (e1) sX e2 > = < One-a-b-1 (e1 sX) e2 >**
(3) **< One-a-b-1 (e1) (eX) e2 > = < One-a-b-1 (e1 (eX)) e2 >**

(we cannot use t-variables in strict Refal). Coming to an end without satisfying the pattern may reqire one more function, which does the job of the remaining sentences. In our case it is simply the end of work. But we must not forget to delete the format parentheses:

(4) **< One-a-b-1 (e1) > = e1**

Thus we have the following translation:

```
One-a-b { eX = < One-a-b-1 () eX >; }

One-a-b-1 {
   (e1) A e2 = e1 B e2;
   (e1) sX e2 = < One-a-b-1 (e1 sX) e2 >;
   (e1) (eX) e2 = < One-a-b-1 (e1 (eX)) e2 >;
   (e1) = e1; }
```

Here all left sides are L-expressions.

When there are repeated e-variables, they are replaced by different variables. Then strict Refal functions for testing equality of expressions must be used.

To illustrate how the extensions of basic Refal, conditions and blocks, are translated into strict Refal, let us take the definition of **Prec-let** (precedence between letters) from Sec.2.7:

```
Prec-let {
   s1 s1 = T;
   s1 s2 where < Alphabet > : eA s1 eB e2 eC
         = T;
   e1 = F; }
```

The execution of a condition $E : L$ is a matching operation. Therefore, we introduce an auxilliary function which will have E as one of its arguments, and will include the matching to L in its definition. All variables defined up to the point where the condition starts (in our case this is **s1** and **s2**) must be included into the arguments of the auxilliary function. The second sentence of **Prec-let** becomes:

< Prec-let s1 s2 > = < Prec-let-1 s1, s2, < Alphabet > >

The definition of the auxilliary function will (originally) consist of two sentences: one for the case when the condition is met,

< Prec-let-1 s1, s2, eA s1 eB e2 eC > = T

(note how necessary **s1** and **s2** are in the argument), and the other when it is not:

< Prec-let-1 s1, s2, eA > = ...

The right side of this sentence must reconstruct the original argument of **Prec-let**, which is **s1 s2** in our problem, (note, again, the necessity for **Prec-let-1** to have **s1** and **s2** in its argument), and call another auxilliary function, which will do the job of **Prec-let** for the remaining sentences:

< Prec-let-1 s1, s2, eA > = < Prec-let-2 s1 s2 >

where

Prec-let-2 { e1 = F; }

Because of the trivial character of **Prec-let-2**, the final definition is simplified:

```
Prec-let {
   s1 s1 = T;
   s1 s2 = < Prec-let-1 s1, s2, < Alphabet > >;
   e1 = F; }

Prec-let-1 {
   s1, s2, eA s1 eB e2 eC = T;
   e1 = F; }
```

It still remains to eliminate the open variables in **Prec-let-1**.

Exercise ... Do that. How many auxilliary functions will be needed?

The case of a block is even easier for translation than that of a condition, because there is no way back from the block. We simply introduce a new function, which, as in the case of a condition, has additional arguments to remember the values of defined variables, but otherwise is defined by the block. Take the function **Merge2** (merge two lists) from Sec. 2.11:

```
Merge2 {
   t1 eX, t2 eY with  < Prec t1 t2 > :
                { T  =  t1 < Merge2 eX, t2 eY >;
                  F  =  t2 < Merge t1 eX, eY >;
                };
   e1, e2  =  e1 e2; }
```

The elimination of the with-statement results in:

```
Merge2 {
   t1 eX, t2 eY  =  < Merge2-1 (t1 eX, t2 eY) < Prec t1 t2 > >;
   e1, e2  =  e1 e2; }

Merge2-1 {
   (t1 eX, t2 eY) T  =  t1 < Merge2 eX, t2 eY >;
   (t1 eX, t2 eY) F  =  t2 < Merge t1 eX, eY >; }
```

To finish the translation into strict Refal, we still have to eliminate t-variables. Our program is usable for sorting sequences of terms which can be both symbols, and parenthesized expressions. We can preserve its full generality; then each of the three sentences which use t-variables will give birth to four sentences. If we assume that we sort either strings of symbols, or lists of expressions, then each of the three sentences will become two. If we know whether it is going to be symbols or parenthesized expressions, we will have the same number of sentences. The list format is, of course, more general: we always can enclose every symbol in parentheses.

3.3 The Generalized Matching Algorithm

By the definition of the operation of matching, the argument must not include free variables. If it does, as is the case in driving, then to execute the operation $E : L$ we must first substitute some values for the variables in E, and then match as usually. Thus the result will depend, generally, on the substituted values. We can, however, analyze the matching $E : L$ with unspecified values of the variables in E and describe what happens under different assumpitons about these values. In this section we define the algorithm of matching generalized in this way. It will be limited to the situations we face when the program is in strict Refal.

3.3.1 MATCHING IN STRICT REFAL

In strict Refal the matching algorithm is much simpler than in the basic version of the language. We only need to consider pairs $E : L$, where E is, as before, an arbitrary object expression, and L is an *L-expression*. Let us call *ultimate terms* such terms of an expression which remain terms after any admissible substitution of values for the expression's variables. Thus **A**, **s1**, **(e1 e2)**, etc. are ultimate terms, while **e1** is not. Since an L-expression has no more than one e-variable on any level of parenthesis structure, we can carry on matching by considering ultimate terms on either end of L, until we either ex-

haust L, if there is no e-variable on the current level, or come to an isolated e-variable, in which case this variable takes on the remaining subexpression of E as its value.

In Chapter 2 we defined matching as a deterministic process by fixing the order in which the terms of the pattern are projected on the argument. We did that in order to define Refal as a deterministic algorithmic language. At this point, however, we are engaged in an analysis of possible results of the operation of the Refal machine. *How* the Refal machine comes to the result of matching is not important for us; we only want to know *what* is the result. Therefore, we can redefine the algorithm of matching (for strict Refal only), so as to leave some freedom in the order of projecting. As we shall see later, this will be very useful for the generalized matching, when E is a general Refal expression.

There is one generalization of the matching operation which we can do at this point. Both parties to the matchig as it was used before were *passive* expressions; they did not include activation brackets. Now we allow activation brackets and will treat them in the same fashion as structure brackets. To a pair of activation brackets in L only a pair in E can correspond in a successful matching. Like structure brackets, activation brackets make pairs where you can jump from one mate to the other, and create closed e-variables. In the context of matching, we can see the expression $<E>$ as **Act**(E), where **Act** is a special symbol which is used only in this role.

Thus the argument E in the matching $E : L$ may be any ground expression, and the L-expression L may include activation brackets.

When the term of L that is being projected is of the form (L_1) or $<L_1>$, and projecting is possible, which means that the term on the corresponding side of E is (E_1) or $<L_1>$, respectively, then in addition to the continuation of matching on the current level, we have a new sub-problem $E_1 : L_1$. Such sub-problems may accumulate; we need, therefore, to maintain a list of current sub-problems. In the algorithm below we use the variable CLASH-LIST to keep the list of subporblems. We use the word *clash* for a matching pair $E : L$ considered as a formal object. For example, if we want to match

(A B C) D E F (I J K) : (e1) e2 F (I e3)

the initial CLASH-LIST is

((A B C) D E F (I J K) : (e1) e2 F (I e3))

If we choose to start projecting from the left end, then after one step it becomes:

(A B C : e1)
(D E F (I J K) : e2 F (I e3))

and then, if we choose to continue on the same level:

(A B C : e1)
(D E F : e2 F)
(I J K : I e3)

We can now choose any of these three sub-problems.

As we go on with matching, the variables of the pattern are assigned some values necessary for a successful matching. Assigning values to free variables is known as *binding*. We shall keep the current binding of the pattern variables as a list of assignments named BIND.

Now we can formulate the algorithm of matching. The notation we use should be understood easily. 'begin' and 'end' are used as brackets enclosing a portion of the algorithm to be executed after 'do'.

variables: CLASH, CLASH-LIST, BIND, as defined above. The letters E, L, and S, possibly with subscripts, stand for some ground expression, some L-expression, and some symbol, respectively. I is some index of a variable.

procedure: to match $E_0 : L_0$, do
begin
Set CLASH-LIST to include one member, which is $E_0 : L_0$;
Set BIND empty;
While CLASH-LIST is not empty, do:
begin (while-loop)
Take any CLASH from CLASH-LIST and remove it;
Use any applicable rule of the following:
case 1. CLASH is *empty* : *empty*. Do nothing.
case 2. CLASH is E : eI. Add E <- eI to BIND.
case 3. CLASH is $SE : SL$ or $ES : LS$. Addt $E : L$ to CLASH-LIST.
case 4a. CLASH is SE : sI L or $ES : L$sI, and there is no assignment to sI in BIND. Add S <- sI to BIND; add $E : L$ to CLASH-LIST.
case 4b. CLASH is SE : sI E or $ES : L$sI, and there is the assignment S <- sI in BIND. Add $E : L$ to CLASH-LIST.
case 5. CLASH is $(E_2)E_1 : (L_2)L_1$ or $E_1(E_2) : L_1(L_2)$ or $<E_2>E_1 : <L_2>L_1$ or $E_1<E_2> : L_1<L_2>$. Add $E_1 : L_1$ and $E_2 : L_2$ to CLASH-LIST.
case 6. None of the above cases is applicable. The matching fails.
end (of while-loop; CLASH-LIST is empty). The matching is completed successfully. BIND is the list of assignments to the variables of L_0.
end (of algorithm).

It is easy to see that because the pattern is an L-expression, the values assigned to its variables in case of a successful matching are unique, without the need to accept any additional convention. Indeed, as long as L is not a single e-variable, an ultimate term on either of its edges must match the term on the same edge of E; our algorithm uses this recursively. This is an important property of strict Refal which does not hold in the basic version.

The other important property, which was referred above as the absence of hidden recursion, is also obvious from our algorithm. The only recursion which is there is a structural recurison on L. If L is given, then the number of elementary operations needed for successful matching can be computed; it does not depend on E, because there is no recursion on E. In the operation of the Refal machine, the left sides of sentences are used as patterns, and the arguments of function calls are arguments in matching. All left sides of sentences are given, of course, when a definite Refal program is given, hence the time necessary for trying the applicability of a sentence is fixed and can be computed. The number of elementary steps needed for an efficient Refal in-

terpreter in order to make the replacement of the left side by the right side is also independent on the view-field and can be computed. Thus the time needed for one step in strict Refal is fixed and does not depend on the contents of the view-field. The number of steps necessary to complete a computation can be a measure of the computation's complexity. This, again, is a property which the basic and extended versions of Refal do not have.

3.3.2 LIST SUBSTITUTIONS

On the basis of the matching algorithm for strict Refal we now proceed to construct an algorithm for generalized mapping E : L, where the expression E is not necessarily a ground expression, but a general expression of Refal. We do not require that it should be an L-expression; the right sides of Refal sentences and the configurations of the Refal machine in metacomputation may have any structure; it is only the left sides that are restricted to L-patterns. But before formulating the algorithm, we must somewhat extend our notation of substitutions.

In the construction of Refal graphs we shall use both simultaneous, and sequential substitutions. We need then to distinguish between these two cases, because the effects of executing the same substitutions simultaneously and sequentially are generally different. For instance, if **eX <- e1** and **e1 <- e2** are executed simultaneously, the new value of **e2** will be the old value of **e1**; if they are executed sequentially in the given order, the new value of **e2** will be the value of **eX.**

We shall call a *varlist* (a short form of 'variable list') an expression of the form

$$(v_1)(v_2) \ldots (v_n)$$

where v_1 etc. are Refal variables, all different. If E is a Refal expression, $\underline{\text{var}}(E)$ is the list of all variables which enter E, e.g.

$$\underline{\text{var}}(\mathbf{e1(A+sX)e1\ eY}) \quad = \quad \mathbf{(e1)(sX)(eY)}$$

We shall treat varlists as unordered sets. We write $V_1 \leq V_2$ to mean that every variable in V_1 is also in V_2. If $V_1 \leq V_2$ and $V_2 \leq V_1$, we shall say that V_1 and V_2 are equal. In programming, it is convenient and efficient to have varlists ordered in some fashion; than two lists will be equal if and only if they are identical as Refal expressions.

The *list contraction*

$$(v_1)(v_2)\ldots(v_n) \text{ -> } (L_1)(L_2)\ldots(L_n)$$

will stand for a simultaneous execution of the contractions $(v_1$ -> $L_1)$ etc. The *list assignment*

$$(E_1)(E_2)\ldots(E_n) \text{ <- } (v_1)(v_2)\ldots(v_n)$$

will stand for a simultaneous execution of the assignments (E_1 <- v_1) etc. As in the case of individual contractions and assignments, the arrows ' -> ' and ' <- ' in list contractions and assignments have exactly the same meaning as the general sign of matching ':', thus to execute the contraction and assignment above, we execute

$$(v_1)(v_2)...(v_n) : (L_1)(L_2)...(L_n)$$

and

$$(E_1)(E_2)...(E_n) : (v_1)(v_2)...(v_n)$$

As we saw in the examples of driving in Sec.3.1, the generalized matching $E : L$ finds such contractions (if any) for the variables in E that after the execution of these contractions on E, it becomes recognizable as L. These contractions must be applied simultaneously. They can be represented with one list contraction

$$\underline{\text{var}}(E) \text{ -> } L^1$$

where L^1 is the list of all the right sides of contractions for individual variables. Consider, e.g., this sentence:

< F (eX A)B eY C D > = eX eY

and suppose we want to drive the function call

< F (e1)s2 e3 >

Then the matching we want to perform is

(e1)s2 e3 : (eX A)B eY C D

Because of the unique matching of parentheses here, the problem immediately reduces to two subproblems: **e1 : eX A**, and **s2 e3 : B eY C D**. The contraction resolving the first subproblem is **e1 -> e1 A**; it requires that **e1** ends with **A**. From the second subproblem we see that **s2** must be **B**, which is expressed by the contraction **s2 -> B**. The remaining part of the target now is **e3**; the remaining part of the pattern is **eY C D**. The contraction necessary in order to make the target to conform to the pattern is **e3 -> e3 C D**. We sum up all this as the list contraction

(e1)(s2)(e3) -> (e1 A)(B)(e3 C D)

We may wish to look at the subclass of the function call under driving which results from the contraction. Then we execute the substitution:

< F (e1)s2 e3 > / ((e1)(s2)(e3) -> (e1 A)(B)(e3 C D))
= < F (e1 A)B e3 C D >

As expected, it is a subclass of the left side of the sentence. We can match it to the left side:

< F (e1 A)B e3 C D> : < F (eX A)B eY CD >

which results in the assignments (the list form):

(e1)(e3) <- (eX)(eY)

In driving, however, we do not need to perform the substitution of contractions. The algorithm which we will formulate below yields for each set of contractions that achieves matching the corresponding assignments for the variables of the pattern, i.e. the left side. Thus the assignments above will come with the completion of the generalized matching.

It is the turn of the replacement now. The replacement is done by substituting the assignments we have found into the right side of the sentence:

((e1)(e3) <- (eX)(eY)) / eX eY
= e1 e3

The final result of the driving is the graph:

Q_1 **((e1)(s2)(e3) -> (e1 A)(B)(e3 C D))** $= Q_2$

(Q_1) **< F (e1)s2 e3 >**
(Q_2) **e1 e3**

In the contractions above, the left-side variable is also among those used in the right side. But we could have introduced a completely new set of variables in contractions:

(e1)(s2)(e3) -> (e8 A)(B)(e9 C D)

or even use the same textually variables as in the sentence:

(e1)(s2)(e3) -> (eX A)(B)(eY C D)

In both cases, as one can easily verify, the resulting graph would be the same. When we use full sets of variables in matchings and replacements, the variables from different sets have no chances to mix.

3.3.3 FACTORIZATION. OLD AND NEW VARIABBELS

In generalized matching we come to the final contraction by steps, each step being a contraction for one variable which is required by the projection of one of the terms of the pattern. We write this contraction as v_i -> L_i, but the true meaning of it is a *list contraction*, where v_i contracts to L_i, and each other variables contracts into itself. For instance, the individual contraction **e1 -> e1 A** in our last example should be understood as

(e1)(s2)(e3) -> (e1 A)(s2)(e3)

Let the sequence of contraction produced by the generalized matching algorithm be:

$$(C_1)(C_2) \ldots (C_n)$$

They must be executed one after another. We can *fold* this sequence into a single list contraction, i.e. compute the composition of the factor-contractions. For that we must know the full initial varlist, V. Then the composition is:

$$V \rightarrow V / C_1 / C_2 / \ldots / C_n$$

The operation of substution / is left-associative; the above should be understood as

$$V \rightarrow (\ldots((V / C_1) / C_2) / \ldots / C_n)$$

The case of assignments is different. The GMA produces individual assignments one after another, but the meaning is that they are parts of the full list assignment, and must be executed simultaneously. Thus if there is no assignment for some variable v_i, this does not mean that this variable retains its old value, but that the value of v_i is yet *undefined*. We simply do not know it yet. To minimize our syntax in examples and in programs, we write list assignments as lists of individual assignments. Unlike the case of contractions,*all* variables defined at the point when the assignment is executed must be present in the list. It must be remembered that the assignments in the list must be performed simultaneously.

It is important to keep in mind that individual contractions are factors of list contractions, because the meaning of an individual contraction may depend on the full list of variables. Take the contraction **e1 -> sX e1**, for instance. If **sX** is not in the current varlist, then this contraction succeeds whenever the value of **e1** starts with any symbol; **sX** is then added to the varlist and takes on this symbol as its value. We refer to such a variable as a *new* variable. If **sX** is already in the varlist, it is referred to as an *old* variable. In this case for the contraction to succeed, the value of **e1** must start with the symbol which is the current value of **sX**, not just any symbol. Without knowing the varlist we cannot execute this contraction or translate it into a computer code. Full list contractions give us all necessary information. If there are no other variables in the varlist, the full contractions for the two cases are:

(new **sX**) **(e1) -> (sX e1)**
(old **sX**) **(e1)(sX) -> (sX e1)(sX)**

The interpretation of list contractions is the same as that of individual contractions (both are matchings): replace the variables in the left side by their values, and match to the right side. Suppose, for example, that in the case of the new **sX** the variable **e1** has the value **A B C**. Then the matching is:

(A B C) : (sX e1)

The result: **sX** is **A**, and **e1** is **BC**. For the case of the old **sX** suppose that **sX** has the value **A**, with the same as before **e1**. The matching then is:

(A B C)(A) : (sX e1)(sX)

It succeeds, with **e1** redefined as **B C**, and **sX**, of course, having the old value. If, however, the value of **sX** is **D**, the corresponding matching

(A B C)(D) : (sX e1)(sX)

fails.

Unlike s-variables, e-variables in L-expressions cannot show up more than once. So, they are always *new*, i.e. can take any subexpression on which they are projected. It is not necessary, however, to give new indexes to e-variables at each contraction. As in the examples above, we can use the left-side e-variable in the right side, if necessary (only once, of course). Such a variable is, strictly speaking, neither old, nor new, but *redefined*. An e-variable can be, also, new. The condition that the value of **e1** must start with a left parenthesis can be expressed by the contraction **e1 -> (eX)e1**, in the assumption that there is no **eX** in the current varlist.

List contractions have the advantage that they can be considered as any other matching, and this can be useful when we write programs for metacomputation. To read them, however, is not so easy as a sequence of individual contractions. It also often happens that a considerable part of a varlist is transformed into itself, then it is irritating an inefficient to write out explicitely those unchanged variables. Thus we shall use full list contractions and assignments mostly as an instrument of a theoretical analysis of pattern operations, and as a guide in programming. In examples, as well as in the data structures of programs, we shall use contractions and assignments in a factorized form, broken into individual contractions.

3.3.4 THE GENERALIZED MAPPING ALGORITHM (GMA)

Structures and variables

Let the clash to resolve be $E_0 : L_0$. A *partial resolution term*, PRT, is a sequence of individual contractions for variables from $\underline{\text{var}}(L_0)$ followed by a list assignment for a subset of $\underline{\text{var}}(L_0)$:

$$(v_1 \text{ -> } L_1)...(v_k \text{ -> } L_k) \mid (E \text{ <- } V)$$

(bars '|' are used to separate elements of structures). A PRT followed by a list of clashes is referred as a PRTC. A STATE is a list of PRTCs, which we write as a sum

$$\text{PRTC}_1 + \text{PRTC}_2 + ... + \text{PRTC}_n$$

Every PRTC is processed by the GMA independently of the others. During the processing, a PRTC term may be eliminated, or give rise to two terms. If the list of clashes in a PRTC becomes empty, we say that the term is *closed* and becomes one of the *resolution terms*, RTs, of the initial clash $E_0:L_0$. The outcome of the GMA is a sum of RTs. An identity operation, i.e. the one like **e1 -> e1**, which is always successful and does nothing, is denoted as **I**. An impossible operation, i.e. the one that always fails, is **Z**. It also represents a sum of zero resolution terms.

Letters E and L, possibly with subscripts, are some expressions and L-expressions, respectively. S is a *syntactic symbol*, i.e. either a symbol, or an s-variable. I stands for some index (of a variable). J' is a *new* index, i.e. such that, at the moment when it is introduced, cannot be found among the variables of $\underline{\text{var}}(E_0)$ or their derivatives.

Auxilliary procedures

To *update* a PRTC by an individual contraction v_i -> L_i, add it to the contraction list and apply it to the left sides of all assignments and clashes in the PRTC. To update a PRTC by an individual assignment, add it to the already existing assignments of the PRTC. To update a PRTC by a sum of individual contractions, take one copy of PRTC for each contraction, update it, and sum the result.

An *internal clash* is a clash $S{:}S'$, where S and S' are either symbols, or s-variables from the same set of variables. To resolve this clash, use the following rules, where A is a symbol:

1. $S : S = \mathbf{I}$
2. $\mathbf{s}I : S = \mathbf{s}I$ -> S
3. $A : \mathbf{s}I = \mathbf{s}I$ -> A
4 If none of the above, **Z**.

Main procedure

begin
Set STATE to consist of one PRTC which is $\mathbf{I} \mid \mathbf{I} \mid (E_0 : L_0)$.
Until all terms in STATE are closed, do:
begin
Pick any CLASH from any PRTC in STATE. Use any applicable rule of the following:
case 1. CLASH is *empty* : *empty*. Delete CLASH in PRTC.
case 2. CLASH is E : $\mathbf{e}I$. Delete CLASH from PRTC, and update PRTC by E <- $\mathbf{e}I$.
case 3. CLASH is $S'E : SL$, or $ES' : LS$. If there is no assignment for S in PRTC, change CLASH to $E : L$, and update PRTC by S' <- S. If S is a symbol S'', or a variable for which there is an assignment S'' <- S in PRTC, then resolve the internal clash $S' : S''$, and let the resolution be R. Change CLASH to $E : L$, and update PRTC by R.
case 4. CLASH is $(E_2)E_1 : (L_2)L_1$ or $E_1(E_2) : L_1(L_2)$ or $<E_2>E_1 : <L_2>L_1$ or $E_1<E_2> : L_1<L_2>$. Change CLASH to $E_1 : L_1$, and add a new clash, $E_2 : L_2$ to PRTC.
case 5L. CLASH is $\mathbf{e}IE : SL$. Update PRTC by $(\mathbf{e}I$ -> $) + (\mathbf{e}I$ -> $\mathbf{s}J'\,\mathbf{e}I)$.
case 5R. CLASH is $E\,\mathbf{e}I : LS$. Update PRTC by $(\mathbf{e}I$ -> $) + (\mathbf{e}I$ -> $\mathbf{e}I\,\mathbf{s}J')$.
case 6L. CLASH is $\mathbf{e}I\,E : (L_2)L_1$. Update PRTC by
$(\mathbf{e}I$ -> $) + (\mathbf{e}I$ -> $(\mathbf{e}J')\mathbf{e}I)$.
case 6R. CLASH is $E\mathbf{e}I : L_1(L_2)$. Update PRTC by
$(\mathbf{e}I$ -> $) + (\mathbf{e}I$ -> $\mathbf{e}I(\mathbf{e}J'))$.
case 7. If none of the above is applicable, delete CPRT from STATE.
end (of processing CLASH)
Output the sum of closed resolution terms. If there are none, output **Z** (matching impossible).
end (of the GMA).

At each step of the GMA we use the necessary and sufficient conditions for the matching to be successful. Therefore, the GMA applied to $E : L$ finds all subclases of E which are, at the same time, subclasses of L. The contractions in the resolution terms resulting from the GMA are represented as sequential individual contractions (where the same variable may be contracted more than once). We can fold them all into one list contraction for the full varlist $\underline{var}(E)$. Then the result of the GMA can be written as:

$$(1) \qquad E : L \;=\; \underline{sum}_{\,i}\,(\underline{var}(E) \text{ -> } L^i)(E^i \text{ <- } \underline{var}(L))$$

where

$$(2) \qquad E \,/\, (\underline{var}(E) \text{ -> } L^i) \;=\; (E^i \text{ <- } \underline{var}(L)) \,/\, L$$

for all resolution terms i. The sum (union) in (1) represents the intersection of the classes E and L. Each subclass (2) can be seen either as the application of the contractions to E, or as the application of the assignments to L. The empty sum (or **Z**) corresponds to the empty intersection, when no subset of E can be recognized as L. If there is only one subset, and the contraction in it is empty (i.e. **I**), then E is a subset of (recognizable as) L. If E is a ground expression, there can be no contractions, and the intersection is either E itself (the matching succeeds), or empty (it fails).

For each resolution term in (1), if any, $\underline{var}(E^i)$ is equal (as a set) to $\underline{var}(L^i)$. This may not me immediately obvious. Of course, the inclusion $\underline{var}(E^i) \leq \underline{var}(L^i)$ must hold according to our rules of operations with full varlists. But why the two are identical in this case?

Imagine that after the i-th subclass of E is identified, we throw away the assignments and perform the matching $E' : L$, where $E' = E \,/\, (\underline{var}(E) \text{ -> } L^i)$; this matching, as we know, must succeed. Matching the expression E' to the pattern L breaks E'into parts which become projections of the elements of L. The elements of L are either constants (symbols or parentheses), or variables. Constants can be projected only on identical constants in E'. Therefore, all variables in E' enter the projections of the variables from L. But then the full list of all the values of $\underline{var}(L)$, i.e. E^i, must include every variable entry in E' and cannot include any other variable: $\underline{var}(E^i) = \underline{var}(E')$. But $\underline{var}(E') = \underline{var}(L^i)$, hence $\underline{var}(E^i) = \underline{var}(L^i)$.

3.3.5 EXAMPLE

Let us illustrate the use of the GMA by the following example. The matching to resolve is:

A s1(C + e2)e2 : A sX(sX eY)eZ sX

The initial PRTC is:

1. **I | I | (A s1(C + e2)e2 : A sX(sX eY)eZ sX)**

Case 3, with S = **A**. Internal clash **A : A**. Its resolution is **I**.

2. **I | I | (s1(C + e2)e2 : sX(sX eY)eZ sX)**

Case 3, with *S* being **sX**, which has no value as yet. Update the assignment list **I** by **s1 <- sX**:

3. **I | (s1 <- sX) | ((C + e2)e2 : (sX eY)eZ sX)**

Case 4. There are two clashes now:

4. **I | (s1 <- sX) | (C + e2 : sX eY)(e2 : eZ sX)**

We peak up the first clash, and will do so below. This is case 3, with *S* being **sX**, which already has a value, namely **s1**. Internal clash **C : s1**. Its resolution is **s1 -> C**. Add it to the contractions and substitute **C** for **s1** in the left sides of the assignments and clashes (**s1** is not to be found in the clashes, though):

5. **(s1 -> C) | (C <- sX) | (+ e2 : eY)(e2 : eZ sX)**

Case 2 (the closed variable **eY**). The clash is resolved:

6. **(s1 -> C) | (C <- sX)(+ e2 <- eY) | (e2 : eZ sX)**

Case 5R. Update by **(e2 ->) + (e2 -> e2 s3)**, where **e3** is a new variable. We now have two PRTCs:

7. **(s1 -> C)(e2 ->) | (C <- sX)(+ <- eY) | (: eZ sX)**
(s1 -> C)(e2 -> e2 s3) | (C <- sX) (+ e2 s3 <- eY) | (e2 s3: eZ sX)

We first go on with the first one. This is case 7, recognition impossible. The first PRTC is eliminated; again, we have one PRTC. It is case 3, where *S* is **sX**, which has the value **C**. The internal clash is **s3 : C**, with the resolution **s3 -> C**. After the update, the PRTC is

8. **(s1 -> C)(e2 -> e2 s3)(s3 -> C) | (C <- sX)(+ e2 C <- eY) | (e2 : eZ)**

Case 2, the closed variable **eZ**. An assignment is added. The clash is resolved. The only PRTC is closed. The last stage of the algorithm yields the resolution term:

9. **(s1 -> C)(e2 -> e2 s3)(s3 -> C) | (C <- sX)(+ e2 C <- eY)(e2 <- eZ)**

The substitution of the contractions in *E* produces:

A C(C + e2 C)e2 C

The substitution of the assignments in *L* produces the same expression -- the intersection of *E* and *L*.

Exercise ... Use the GMA to resolve the clash:

e1 + e2 : s1(e2)e3 s4 A

Find an object expression which belongs to two subclasses of the resolution. Find an expression belonging to all the subclasses.

3.3.6 ORTHOGONALITY OF CONTRACTIONS FROM GMA

As one can see from Exercise ..., the subclasses resulting from the GMA may overlap. But if E is also an L-expression, then the subclasses will be disjoint. This is a special case of a more general (and more important) property of the contractions generated in the GMA: when contractions from different resolution terms are substituted into *any* L-expression L', not necessarily the argument of the matching, they produce disjoint subclasses of L'. We shall call such contractions *orthogonal*.

THEOREM. Let $E : L$, where E is a ground expression, and L an L-expression, be resolved by the GMA as

$$E : L \;=\; \underline{\mathrm{sum}}_i \, C^i A^i$$

where C^i are contractions for $\underline{\mathrm{var}}(E)$, and A^i assignments for $\underline{\mathrm{var}}(L)$. Let L' be an L-expression, and $\underline{\mathrm{var}}(E) \leq \underline{\mathrm{var}}(L')$. Then L' / C^i are L-classes disjoint for different i. (If $\underline{\mathrm{var}}(E) < \underline{\mathrm{var}}(L')$, then C^i must be extended by including identical contractions for the variables in L' but not in E).

Proof. In the contractions for e-variables required by the GMA, e-variables can generate another expression variable only confined in parentheses (cases 6L and 6R). This variable is always *new*, thus excluding the possibility of repeated e-variables after the subsitution. Therefore the expression resulting from substitution will remain an L-expression if it was an L-expression.

More than one subclass can be produced in the GMA only by a substitution for an e-variable in cases 5L, 5R, 6L, 6R. In all these cases the first subclass eliminates the e-variable, while in the second subclass the replacement for the e-variable has at least one ultimate term. Suppose that there is an e-variable on the top level of structure in L'. There can be no more than one e-variable on any level in L'. Therefore, all other syntactic terms on the top level are ultimate terms. If the number of ultimate terms in L' before the substituion was n, then in the first subclass, and in all subclasses which can be produced by further contractions, the number of ultimate terms on the top level will be exactly n, because there is no e-variable on this level any more. In the second subclass, and in all subclasses produced by further contractions the number of ultimate terms will be at least $n + 1$. Therefore, no object expression can simultaneously belong to both groups.

The further proof is by structural induction on the e-variables of L'. Pick some e-variable on the level k in L'. If $k = 0$, then substitutions for $\mathbf{e}I$ can produce only disjoint subclasses, as we have just proven. Thus we assume that $k \geq 1$. Let the subterm containing $\mathbf{e}I$ be

(1) $\qquad (E_1 \; \mathbf{e}I \; E_2)$

Suppose that the following proposition is true for $\mathbf{e}I$: all e-variables on the levels higher than $\mathbf{e}I$ (i.e. k-1 or less) can produce, in substitutions from GMA, only disjoint subclasses. Then for L' to produce two overlapping (not disjoint)

subclasses all the substitutions for e-variables on the levels $\leq k$-1 must be the same, and then there are two possibilities: (a) the terms produced from (1) by some pair of substitutions for **e***I* are overlapping, (b) the same substitution is used in (1), and another e-variable produces the desired result. However, the possibility (a) must be excluded, because it would mean that two substituions for an e-variable on the top level in

(1') E_1 **e***I* E_2

produced overlapping subclasses, and this is impossible. Therefore, another e-variable, not **e***I*, must be responsible. If we start with an e-variable on the top level in L', then the inductive assumption will be true in the beginning, and the search for a suitable variable will end without finding one.

3.4 Step-long function transformations

Using the generalized matching algorithm we can perform a few transformations of function definitions which, as it is easy to see, leave the function strictly equivalent to what it has been. The transformations we consider in this section are markedly different from the transfrmation by metacomputation in that they do not trace the work of the Refal machine for more than one step. If **F2** is a function obtained by such a transformation from **F1**, then for every argument E the result of one step with **< F2** E **>** in the view-field will be exactly the same as with **< F1** E **>**.

1. *The Transposition rule.* If two adjacent sentences in a function definition

```
...
L1 = R1;
L2 = R2;
...
```

have *orthogonal* left sides L_1 and L_2, by which we mean that the classes represented by them are disjoint, then we can transpose these sentences:

```
...
L2 = R2;
L1 = R1;
...
```

In order to establish that L_1 and L_2 are orthogonal, we match $L_1 : L_2$ (or $L_2 : L_1$), and check that the resolution is **Z** (matching impossible).

2. *The Screening rule.* If there are two sentences anywhere in a function definition

$$
\begin{array}{l}
\ldots \\
L_1 \;=\; R_1; \\
\ldots \\
L_2 \;=\; R_2; \\
\ldots
\end{array}
$$

and the left side L_1 of the first sentence *subsumes* the left side L_2 of the second sentence, by which we mean that the class L_2 is a subset of L_1, then the second sentence can be eliminated; we shall say that it is *screened* by the first sentence. To establish that L_1 subsumes L_2, we match $L_2 : L_1$, and check that the resolution exists and there are no contractions in it.

3. *The Subsumption rule.* If there are two adjacent sentences in a function definition

$$
\begin{array}{l}
\ldots \\
L_1 \;=\; R_1; \\
L_2 \;=\; R_2; \\
\ldots
\end{array}
$$

and there exists an assignment S for the variables in L_2 such that

$$
\begin{array}{l}
S/L_2 \;=\; L_1 \\
S/R_2 \;=\; R_1
\end{array}
$$

then the first sentence can be eliminated. It is subsumed by the second sentence. To establish that this situation takes place, we match $L_1 : L_2$ and check that the matching occurs without contractions. Then we take the assignment resulting from the matching, apply it to R_2 and compare the result with R_1.

These transformation rules do not cover all improvements which can be made in a program without altering the result of every step. Consider this program:

```
F { s1 e2  = R1;
    (e1)e2 = R2;
     = R3;
    e1 = R4;
  }
```

The fourth sentence will never be applied, because every expression either starts with a symbol, or starts with a left parenthesis, or is empty. Thus it can be deleted. Yet it is not possible to do this using only the three transformation rules above. We shall see in Chapter ... how this transformation can be accomplished in metacomputation.

3.5 The Algorithm of Driving

We can formulate now the general algorithm of driving. First we do it for the inside-out evaluation order. We want to construct the graph of states for a configuration Q. Let P be the primary active subexpression in Q. It has the form $<F\ E>$, where F is a function symbol, and E a passive expression. Q can be decomposed as follows:

$$Q \;=\; (P \leftarrow \mathbf{eX}) / Q' \;=\; (<F\,E> \leftarrow \mathbf{eX}) / Q'$$

where we assume that **eX** is not among the variables used in Q. The variables like **eX**, which are used for separating a subexpression from a larger expression, will be referred to as *liaison* variables.

Let the sentences for F be:

$$<F\,L_1> \;=\; R_1$$
$$<F\,L_2> \;=\; R_2$$
$$\ldots$$
$$<F\,L_n> \;=\; R_n$$

Imagine that Q is in the view-field of the Refal machine, and trace its evolution. For those values of the free variables in Q with which E matches L_1, the Refal machine will use the first sentence to make a step. It is a certain subclass of E. To find it, we resolve, using the GMA, the clash:

$$E : L_1 \;=\; \underline{\mathrm{sum}}_k\, (\underline{\mathrm{var}}(E) \rightarrow L_1^k)(E_1^k \leftarrow \underline{\mathrm{var}}(L_1)), \qquad k = 1, 2, \ldots M_1$$

Under each contraction in the sum, the Refal machine will use the first sentence for F and replace P by

$$P_t^k \;=\; (E_1^k \leftarrow \underline{\mathrm{var}}(L_1)) / R_1$$

because the assignment part of the resolution gives us the values taken by the variables of L_1 in the process of matching. It is only the variables from L_1 that are allowed to be used in R_1, hence after the substitution we have an expression that depends only on the variables of E and its derivatives.

Thus the first part of the graph of states for Q, which corresponds to the first sentence in the definition of F, will consist of M_1 branches:

$$\{\ (\underline{\mathrm{var}}(E) \rightarrow L_1^1) \;=\; (P_1^1 \leftarrow \mathbf{eX}) / Q'$$
$$+\ (\underline{\mathrm{var}}(E) \rightarrow L_1^2) \;=\; (P_1^2 \leftarrow \mathbf{eX}) / Q'$$
$$\ldots$$
$$+\ (\underline{\mathrm{var}}(E) \rightarrow L^{M1}{}_1) \;=\; (P^{M1}{}_1 \leftarrow \mathbf{eX}) / Q'$$
$$\ldots$$

For those members (ground expressions) of the primary configuration P which do not belong to any of the subclasses separated in the matching $E : L_1$, the first sentence will be found unapplicable. The Refal machine will then try to apply the second sentence, which we should take into account by separating another group of subclasses of E and adding it to the first group.

Repeating this procedure for each sentence in the definition of F, we come to a graph which ends with the branch for the last subclass separated in the use of the last sentence:

```
...
+ (var(E) -> L^Mn_n)  =  (P^Mn_n <- eX) / Q'
}
```

The subgraph following a configuration Q in a graph will be referred to as the *development* of Q, and denoted as $\underline{\text{dev}}(Q)$.

There is an obvious optimization to the construction of the graph of states as described above. If for the i-th sentence of F the argument E is found to match L_i without contractions, which means that E is a subclass of L_i, then the branches originating from all the sentences starting with the $i+1$-st can be omitted, because they will never be used.

Let us trace the dynamics of varlists in the process of driving. As we know, the only operation of Refal is the operation of matching. We shall insert between the designations of matchings (clashes), and possibly at the beginning and end, the full lists of variables that are defined at that point. They will be enclosed in square brackets, and viewed as comments. Recall how a generalized matching $E : L$ is executed. First the values of the variables are substiuted in E, then the usual matching is performed. Therefore, each variable found in E must be in the current varlist; it is not necessary, though, that each variable from the varlist is actually used in E. After the matching the new set of variables is defined, $\underline{\text{var}}(L)$; they, *and only they*, are defined after the matching. Thus we have the following scheme:

$$[V_1]\,(E : L)\,[V_2] \qquad \text{where } \underline{\text{var}}(E) \le V_1, \text{ and } \underline{\text{var}}(L) = V_2$$

For contractions and assignments we have the schemes:

$$[V_1]\,(V_1 \text{ -> } L)\,[\underline{\text{var}}(L)]$$
$$[V_1]\,(E \text{ <- } V_2)\,[V_2] \qquad \text{where } \underline{\text{var}}(E) \le V_1$$

When we construct a graph of states, the variables in the initial and subsequent configurations will be referred to as *C-variables* (configuration variables); the variables in the sentences of the program will be called *P-variables* (program variables). The scheme of one step of driving through a sentence $<L^P> = R^P$ is as follows:

Q_1	The initial configuration.
$[V^C]$	$= \underline{\text{var}}(Q_1)$. The C-variables of the initial configuration.
$(V^C \text{ -> } L)$	Contractions of C-variables necessary to match Q_1 to the left side L^P; they result from the GMA.
$[V^{C'}]$	$= \underline{\text{var}}(L)$. A new set of C-variables derived from V^C.
$(E \text{ <- } V^P)$	where $V^P = \underline{\text{var}}(L^P)$. This is the set of assignments resulting from the GMA.
Q_2	$= (E \text{ <- } V^P) / R^P$. The configuration resulting from driving.
$[\underline{\text{var}}(Q_2)]$	$\le [V^{C'}]$.

(The reason for the last relation: since $\underline{\mathrm{var}}(R^p) \leq \underline{\mathrm{var}}(L^p) = V^p$, and the values E assigned to V^p depend only on V^c', the substitution $(E <- V^p) / R^p$ produces an expression that may depend only on V^c'; therefore, $\underline{\mathrm{var}}(Q_2) \leq [V^c']$). We start from a configuration depending on V^c, and end up with a configuration depending on the derived set V^c'. The P-variables used in the step are gone.

Now we shall define the outside-in driving, which is our main mode of driving. If in the inside-out driving we take up an active term of the configuration Q which contains no other active terms, in the outside-in driving we start with an active term which is not a part of any other active term of Q. If there are more than one such terms, we can pick up any of them: as we know, parallel processes do not interact in strict Refal. We, again, decompose the configuration:

$$Q = (P <- \mathbf{eX}) / Q' = (<F\,E> <- \mathbf{eX}) / Q'$$

where P is the active term we have chosen for driving. As in the inside-out computation or driving, we call P a primary active term. However, E is not necessarily a passive expression any more, but may include active subexpressions. This may prevent us from completing the matching and force to suspend the evaluation of P and take a subexpression of P as a new primary active term for evaluation. When we do matching for the purpose of outside-in driving, we must distinguish between two kinds of failure. If we take, e.g., the clash **(A) : sX**, we can positively say that the matching is impossible. For the clash **< F e1 > : sX**, the matching is again impossible if we consider **< F e1 >** as a fixed stage of the Refal process represented by this active expression. But this is not the approach we want in driving where we look forward to the completion of evaluation processes. When **< F e1 >** is computed, it still may turn to be a single symbol, which will make the matching possible.

As long as active terms do not show up on either edge of the argument E in the matching $E : L$, the GMA simply does not notice them; for subclasses where the matching is possible, active terms, if any, will enter the values of e-variables. When an active term $<E'>$ appears in a position where it must me matched to a term in L, we cannot successfully complete the matching without first computing (at least, partially) $<E'>$. We shall say that this term *hinders* the process of matching. We want to separate such cases from the cases of certain failure. The following additions must be made in the GMA for cases 3 and 4:

case 3'. CLASH is $<E'>E : SL$ or $E<E'> : LS$. The term $<E'>$ hinders successful matching.

case 4'. CLASH is $<E'>E : (L_2)L_1$ or $E<E'> : L_1(L_2)$. The term $<E'>$ hinders successful matching.

Even though we cannot successfully complete the matching in case of a hindrance, we may be able to establish that the matching fails no matter what is the result of the evaluation of the hindrance. Take this clash, for instance:

(1) **< F e1 > e2 A B : sX eY C**

The term **< F e1 >** hinders successful matching. However, by turning to the right end of the pattern we establish, without evaluating this term, that the matching will always fail. The situation on the other end of the pattern is not the only potential cause of a failure. Remember that PRTC's in the GMA include, generally, a number of clashes, which all must be successfully resolved for the success of the original matching $E_0 : L_0$. In this matching:

(2) **(< F e1 >)A : (sX)B**

we try to resolve **< F e1 > : sX**, and cannot do that because of a hindrance, but the other clash, **A : B**, causes a failure anyway.

Thus the following way of treating hindrances can be adopted:

In case of a hindrance in a PRTC, suspend the matching on the current side of the current clash, and continue the matching on the other end and in the other clashes. If progress is nowhere possible because of hindrances, take any of them as a new primary active term.

We will make no error, though, if we change the primary active term the first time that we meet a hindrance. If the matching is impossible, we will discover this in due course anyway. And to find that the matching is possible, all hindrances must be first evaluated. In the assumption that the initial expression we put in the view-field of the Refal machine is a defined and finite process, the choice of the primary active subexpression at each step of metacomputation determines only the order of operations, but not the final result.

The stipulation that the overall process is defined and finite is very important, however. Take the matching (1) above. We asserted that it will fail. But if the function **F** is undefined, the Refal machine will come to an abnormal stop trying evaluate it. If **< F** E **>** for some E is an infinite process, the Refal machine will run forever. In both cases, the matching will simply have no chance to fail if we do not dismiss the true, i.e. inside-out, semantics of Refal. This brings us to the question: what kind of equivalence do we guarantee between the original Refal program and the program resulting from outside-in driving?

Let the initial configuration be Q, and let us take a binding for the variables of Q described by the assignment (E <- $\underline{\text{var}}(Q)$), such that Q with this binding is a defined finite process. Then with every active term in Q we can associate its final stage. The Refal machine, when it evaluates a function call, first computes and replaces by their final stages all the active terms in the argument. When we construct a graph of states by the outside-in driving, we assume that all active terms in the argument stand for some expressions -- which is true -- and proceed as far as possible without knowing what these expressions are. We compute active terms in the argument of a function call only when their final stages make impact on the computation of the function. Otherwise, the construction of the graph mimicks the operation of the Refal machine. Therefore, the final stage of Q as given by the graph of states will be the same as computed by the inside-out Refal machine. **If the inside-out computation leads to a definite result, then the outside-in computation leads to the same result.**

Now let us allow for the possibility that some sub-processes in Q may not terminate successfully. With outside-in evaluation, an active subexpressions may be left uncomputed if the overall configuration does not depend on its final stage. If the active term the computation of which is canceled stands for an infinite or undefined process, then the outside-in computation may lead to a definite result even though the original Refal program understood according to the inside-out semantics does not. Outside-in driving may extend the domains where the functions involved are defined.

It is also worth noting that because of the cancellation of some unnecessary computations, the outside-in program may be much more efficient than the original inside-out program.

We shall say that a program P' *covers* a program P if any ground expression which initiates in the Refal machine a finite process under the program P, does so also under P', and the final stages of these processes are the same. Outside-in driving transforms a program P into a program P' which covers P. The relation of covering, as one can easily see, is reflexive and transitive.

A few examples. Consider this program:

```
* Predicate F1 checks whether the argument starts with A.
F1 {A e1 = T;
    e1 = F; }
* This function converts symbols in a string, one by one.
F2 {A e1 = B <F2 e1>;
    B e1 = A <F2 e1>;
    s2 e1 = s2 <F2 e1>;
     = ; }
* The composition of the two functions
F { e1 = <F1 <F2 e1> >; }
```

Since **F1** depends only on the first symbol of its argument, the computation of the whole value of **F2** by **F** is unnecessary. Driving eliminates this inefficiency. It yields the following definition of **F**:

```
F {A e1 = F;
   B e1 = T;
   s2 e1 = F; }
```

This definition can be further improved. We first use the Transposition rule (see Sec. 3.4), for the first two sentences:

```
F {B e1 = T;
   A e1 = F;
   s2 e1 = F; }
```

Then we use the Subsumption rule to eliminate the second sentence:

```
F {B e1 = T;
   s2 e1 = F; }
```

Let us redefine **F2** as follows:

```
F2 {s1 e2  =  s1 <F2 s1>; }
```

Now **<F2 e1>** and, therefore, **<F e1>** which calls it, are defined nowhere; the domains of both functions are empty. But if we transform the definition of **F** by the outside-in driving, we get the following definition:

```
F {A e1  =  T;
   s2 e1  =  F; }
```

which defines **<F e1>** for every string **e1** that starts with a symbol. We can see the "final stage" of **<F s1 e2>** as the infinite string

s1 s1 s1 s1 s1 ... etc.

The function **F1** checks whether the first symbol in this string is **A**, and completes computation in one step.

In this way we can use infinite processes and think about them as constructing "infinite objects". Such processes can be used in conjunction with functions that analyze these objects in a manner co-ordinated with the manner of their construction. We call these functions as if their arguments were really infinite objects, and then transform the program by driving.

Suppose, e.g., that we have a predicate **P**, and want to find the minimal whole number for which **P** holds. We define the left-to-right generator of whole numbers **<Num 0>**, where **Num** is defined as

```
Num {sN  =  sN <Num <Add sN, 1> >; }
```

and the function **Look-p**:

```
Look-p {
   s1 e2 & <P s1>: { T  =  s1;
                     F  =  <Look-p e2>;
                   };
      }
```

Then we metacompute **<Look-p <Num 0> >** using the outside-in driving. We leave this computation to the reader as **Exercise ...** (The answer, as always, is in the end of the book). If we want to find the first *prime* number satisfying **P**, we define a generator of prime numbers **<Prime-num 2>** and metacompute **<Look-p <Prime-num 2> >**.

One could be disappointed that our basic method of function transformation does not lead to a strictly equivalent function, but only to a function that *covers* the original one. In fact, however, this is the strong point of the method rather than the weak one. As we have just seen, it allows us to deal with processes generating infinite objects. More generally, optimization of an algorithm is closely tied to the extension of its domain. Whenever we avoid some part of computation as unnecessary, or simplify it some other way, we potentially eliminate the restrictions necessary for successful completion of that computation, and this may bring new elements into the domain. This feature of optimization goes beyond outside-in driving. Consider the following function:

```
Bits { 0 eX  =  0 <Bits eX>;
       1 eX  =  1 <Bits eX>;
         =  ; }
```

The domain of **Bits** is the set of all binary numbers, and its value is identical to its argument. The obvious optimization (which cannot be accomplished by simple driving) is to redefine **Bits** as the identity function:

```
Bits { eX  =  eX; }
```

The new funcion, however, is defined on every expression, not only on binary numbers. If we stick to the reqirement of strict equivalence to the original function, we have to check that the argument consists only of **0**s and **1**s. But then say good-bye to optimization.

3.6 Transformation of Refal graphs

3.6.1 REFAL GRAPHS AS PROGRAMS

We constructed Refal graphs as generalized histories of the evaluation of general expressions (configurations) placed in the view-field of the Refal machine. A finished Refal graph, however, is also a program for the evaluation of any ground expression which belongs to the initial configuration. As objects of manilulation, Refal graphs have certain advantages over Refal programs. The format of Refal programs is suitable for defining and discussing algorithms, but we shall use Refal graphs as our basic language in metacomputation. In this section we make some additions to our notation of graphs and formulate the rules of transformation preserving the meaning of the graph as a program. Driving, in particular, will be represented as a graph transformation.

We shall use in Refal graphs a special variable, namely **e0**. It can be described as the variable for input/output operations. Its value is the current contents of the view-field of the Refal machine. The graphs we constructed before started and ended with configurations. Now we modify slightly the format of the graphs. The initial configuration Q_1 of a graph will now be represented as the contraction **e0** -> Q_1, which should be read: "if the view-field is Q_1, then ...". The final configuration Q_2 on every branch of the graph will be replaced by the assignment Q_2 <- **e0**, to be read: "put Q_2 in the view-field". Such assignments correspond to the nodes of the Refal graph in the old notation and will be often referred to as nodes. By introducing **e0** we eliminate configurations as distinct syntactical elements of a graph, and express Refal graphs as trees constructed from matching operations, of which we already know two varieties, contractions and assignments, and will soon add one more -- *restrictions*.

The context-free syntax of Refal graphs is as follows. A graph is one of the following:

a *walk*, followed by a *walk-end*;
a *graph-sum* enclosed in braces;
a walk followed by a graph-sum in braces.

A graph-sum is one of the following:

empty;

a single graph;

$G_1 + ... + G_n$, where G_1, etc. are graphs.

A walk is a sequence of *matching operations*.

A walk-end is an assignment to **e0**.

A matching operation is one of the following:

a list contraction (V -> L);

a restriction (# G), where G is a *contraction graph*, i.e. a graph-sum where all walks consist only of contractions (no restrictions and no walk-ends); walks in contraction graphs can be factorized, as in other graphs; (# G) will be referred to as the *negation* of G;

a (passive) list assignment (E <- V);

a *subgraph* assignment (or active assignment) ({G} <- **e***I*), where {G} is a graph, which in this context is referred to as a *subgraph*, and **e***I* is referred to as the *liaison variable*.

a *define* operation def(E <- v), where v is a variable;

a *delete* operation del(v).

There is an additional condition a walk must meet in order to be syntactically valid, which will be referred to as *varlist coupling*. It must be possible to associate with every walk its *input* and *output varlists*, and with a graph its *input varlist*, while observing certain rules. For the basic operations these rules have already been partly discussed. If we put the input and output varlists in square brackets before and after the walks, and the input varlist of a subgraph just before the subgraph, the conditions to be met are as follows:

$[V]\ (V \text{ -> } L)\ [\underline{\text{var}}(L)]$

$[V]\ (\#\ V \text{ -> } L)\ [V]$

$[V]\ (E \text{ <- } V')\ [V']$ where $\underline{\text{var}}(E) \leq V$

$[V]\ ([V_g]\{G\} \text{ <- } \mathbf{e}I)\ [V(\mathbf{e}I)]$ where $V_g \leq V$

$[V]\ \underline{\text{def}}(E \text{ <- } v)\ [V(v)]$

$[V_1(v)V_2]\ \underline{\text{del}}(v)\ [V_1V_2]$

The rules of combining operations are:

1. To sum two or more graphs $G_1 + G_2 + ...$, there must be a varlist V which satisfies the conditions for being the input varlist of all the graphs G_1 etc. Then V becomes the input varlist of the sum.
2. To concatenate W_1 and G_2, where W_1 is a walk, and G_2 a graph (or a walk, in particular), there must be a varlist V which serves both as the output for W_1, and the input for G_2.

The reason for these requirements is the interpretation of a Refal graph as a program. We have already seen in examples how a Refal graph can be interpreted as a program and converted into a program. We give now a formal definition of the interpretation of a Refal graph.

While the Refal machine operates on the view-field, a Refal graph works with *environments* as data structures. An environment is a binding where all variables of the current varlist are assigned some *ground* expressions as their values; we represent environments as list assignments E^{gr} <- V. To interpret a Refal graph, we must be given the initial environment. Then with every point in a graph we can associate a definite varlist. Indeed, the rules above

have only one indeterminacy in defining input and output varlists: the input varlist of an assignment must include all variables in var(E), but may also include some other variables. If the graph starts with an assignment, its input varlist will be the varlist of the initial environment. For an assignment which is not at the start, we take the output of the preceding operation as the input varlist. The output varlist of a walk is uniquely defined by our rules as the output varlist of the last matching operation. Now the execution of the graph as a program will be a walk through the graph with the assigning of certain values to the varlists along the path. This is done according to the following rules.

1. To execute the contraction (V -> L) with the input environment E^{gr} <- V, execute the matching $E^{gr} : L$. If it fails, *backtrack* (see p.5). If it succeeds, the assignment to var(L) is the output evironment.
2. To execute the restriction (# V -> L) with the input environment E^{gr} <- V, execute the matching $E^{gr} : L$. If it succeeds, *backtrack*. If it fails, the unchanged input environment becomes also the output environment. To execute the negation of a sum of contractions, execute sequentially the negations of the components.
3. To execute the assignment (E <- V') with the input environment E^{gr} <- V, take ((E^{gr} <- V) / E <- V') as the output environment.
4. To execute the subgraph assignment ({G} <- **e***I*) with the input environment E^{gr} <- V, first execute the subgraph {G} with this environment; let its output environment be (E <- **e0**). Take ($E^{gr}(E)$ <- V(**e***I*)) as the output environment for the assignment. Thus, all the variables in V retain their values, and a new variable **e***I* appears, with the value resulting from the execution of the subgraph.
5. To execute def(E <- v) add v to the current varlist and assign E to it. To execute del(v) exclude the variable v from the current varlist.
6. To execute a walk, execute its constituend operations sequentially. To execute a graph-sum, start with the execution of its first graph-term. If you come to a walk-end, the execution is successfully terminated. If you have to *backtrack*, eliminate the current graph-term, restore the environment at the beginning of it, and start executing the next graph-term. If the current graph-term is the last in the sum, perforem the same operation on the next level up the tree structure, etc. If you have to backtrack from the last graph-term on the top level, the execution fails and the result is undefined. This corresponds to the abnormal stop of the Refal machine.

It follows from this definition of graph interpretation that all matching operations are distributive with respect to addition:

$$W(G_1 + G_2) = WG_1 + WG_2$$

The values of variables in an environment are ground, but not necessarily *object* expressions (because execution of a graph simulates the processes in the view-field of the Refal machine, and they involve manipulation of activation brackets). Therefore, the execution of contracitons may stumble over a *hindrance*. In this case we shall say that the graph is *unexecutable*. A graph is *executable* if with any environment involving only object expressions it can be executed without hindrances.

It is easy to give an example of an unexecutable graph:

```
(< Fab e1 > <- e2) {(e2 -> B e2)(C < Fbc e2 > <- e0)
                   + ...
                   }
```

The execution of the first branch meets a hindrance: **e2** has an active value, and the contraction cannot be executed. However, this example is, in a sense, artificial. Graphs resulting from metacomputation are always executable by their construction. Here we again face the double nature of a Refal graph: it is both a program for executing a computation with specific input data, and a family of computational histories constructed in metacomputation. On each branch of the Refal graph emerging in metacomputation the appearance of a hindrance causes a new choice of the primary active term; hindrances in the final product are not tolerated. The example above is taken from Sec. 1, where we drive the expression **< Fbc < Fab e1 > >**. At the beginning we try to drive **Fbc** with the active environment (**< Fab e1 > <- e2**), but because of the hindrance we switch to **Fab**, and the hindrance is eliminated.

The following two aspects of the execution of Refal graphs must be kept in mind.

1. A Refal graph is a tree, not a general graph; its execution corresponds to the execution of a finite number of steps of the Refal machine, even though this number may be different on different branches. Because of that, the problem whether a given graph is executable is easily solvable: we take a generalized initial environment where all variables have arbitrary values, and execute the graph using the generalized matching; this is, essentially, re-driving of the graph. If on no branch we encounter a hindrance, the graph is executable.
2. When it is necessary to evaluate an expression with nested activation brackets -- or rather to make a number of evaluation steps -- a Refal graph defines specifically which active subexpression must be taken as primary at each step. In constructing a Refal graph we are free to use inside-out or outside-in evaluation order, or just picking up primary terms arbitrarily; such decisions can be taken for any branch independently of others.

3.6.2 RESTRICTIONS

Consider a function definition in the general form:

$$
\begin{aligned}
&\langle F\, L_1 \rangle = R_1;\\
&\langle F\, L_2 \rangle = R_2;\\
&\ldots\\
&\langle F\, L_n \rangle = R_n;
\end{aligned}
$$

The corresponding Refal graph, which can be obtained by driving **< F e1 >**, is

$$
\begin{aligned}
&\textbf{(e0 -> } \langle F\ \textbf{e1} \rangle \textbf{)}\ \{(\textbf{e1 -> } L_1)\ (R_1 \textbf{ <- e0})\\
&\qquad + (\textbf{e1 -> } L_2)\ (R_2 \textbf{ <- e0})\\
&\qquad \ldots\\
&\qquad + (\textbf{e1 -> } L_n)\ (R_n \textbf{ <- e0})\\
&\qquad \}
\end{aligned}
$$

Because branches are used sequentially, they are not completely independent: the actual use of a branch in a graph is limited to those environments (i.e. exact states of the Refal machine) for which all preceding branches originating at the same node were found unapplicable. We introduce *restrictions* in order to achieve a complete independence of branches. Restrictions sieve out those exact states which will not actually reach the branch.

The simplest way to incorporate restrictions into the graph above is as follows:

$$
\begin{aligned}
(\mathbf{e0} \rightarrow \langle F\ \mathbf{e1}\rangle)\ \{&(\mathbf{e1} \rightarrow L_1)\ (R_1 \leftarrow \mathbf{e0})\\
&+ (\#\ \mathbf{e1} \rightarrow L_1)\ (\mathbf{e1} \rightarrow L_2)\ (R_2 \leftarrow \mathbf{e0})\\
&+ (\#\ \mathbf{e1} \rightarrow L_1)(\#\ \mathbf{e1} \rightarrow L_2)\ (\mathbf{e1} \rightarrow L_3)\ (R_3 \leftarrow \mathbf{e0})\\
&\ldots\\
&+ (\#\ \mathbf{e1} \rightarrow L_1)(\#\ \mathbf{e1} \rightarrow L_2)\ldots(\#\ \mathbf{e1} \rightarrow L_{n-1})\\
&\qquad\qquad (\mathbf{e1} \rightarrow L_n)\ (R_n \leftarrow \mathbf{e0})\\
\}&
\end{aligned}
$$

We shall refer to such a graph as *complete with restrictions*. In each branch there is a check that all preceding branches are unapplicable. A graph which is complete with restrictions has the pleasant property that the branches outcoming from a node can be transposed and put in any order. We shall see later, however, that there are good reasons not to change the order of branches, even though they may be complete with restrictions. But first we want to make an improvement in the graph above.

Consider the second branch, which is the first branch including a restriction. In the execution of the graph it is first checked that **e1** does not contract to L_1, and then it contracts to L_2. This may result in unnecessary operations. Take the case when L_2 is orthogonal to L_1, i.e. their intersection is empty. Then any value of **e1** which belongs to L_2 certainly does not belong to L_1; the restriction (# **e1** -> L_1) will be, therefore, always satisfied, and there is no need to check it in order to determine whether the sentence will be used. Speaking in terms of sets of values of **e1**, we identify the set corresponding to the second sentence as the difference between the classes L_2 and L_1. But if these classes are disjoint, then the difference is L_2.

This inefficiency will be eliminated if we first execute the contraction, which is necessary anyhow, and then check whether the values assigned to $\underline{\text{var}}(L_2)$ are such that the argument did not match one of the preceding sentences. We want a *commutation relation* which would transform the combination Rectriction-Contraction (RC for short) into an equivalent reversed combination CR. This relation can be established by the following reasoning. Suppose a ground expression passed successfully the contraction to L_2. Since it must not, at the same time, belong to L_1, the restriction we are looking for must eliminate the members of the intersection L_2 $\underline{\text{int}}$ L_1. This intersection must be represented by contractions to the variables of L_2, because the restrictions we want are restrictions on $\underline{\text{var}}(L_2)$. Therefore, we match $L_2 : L_1$, throw away assignments, and retain only the contractions. Let this procedure be denoted as $\underline{\text{contr}}$; it gives us a sum of contractions for $\underline{\text{var}}(L_2)$, and we convert them into restrictions. The desired commutation relation can be written as:

$$
\text{(RC)} \qquad (\#\ V \rightarrow L_1)(V \rightarrow L_2) \;=\; (V \rightarrow L_2)(\#\ \underline{\text{contr}}(L_2 : L_1))
$$

The sum of contractions, when it is used as restriction, is executed sequentially, and in case if any of the terms in the sum turns to be applicable the execution comes to the prohibitive stop. If all the terms are found unapplicable, this is the "no objections" result. We have, therefore, the relation:

(RR) $\quad (\#\,(V \to L_1) + \ldots + (V \to L_n)) = (\#\,V \to L_1) \ldots (\#\,V \to L_n)$

Since restrictions do not change the environment, they commute, and so do the branches in contraction graphs, even though the branches of a graph of states generally do not. It is easy to see whence the difference. When a sum of walks is used as a program and two walks can be applicable to the same expression under evaluation, the result of replacement may be different because the assignments are different. When a sum of walks is used as a restriction, there are no assignments in the graph, and it is only the applicability of one of the contraction that matters.

Using the commutation relation (RC) and the composition relation (RR), we can transform every walk in the function definition to what we shall call the *normal form*: the sequence CRA, i.e. contraction, restriction, assignment (the *normal order*). Take, for example, the third sentence, which includes two restrictions: R_1R_2CA. We first combine R_1 and R_2 into one restriction using, in the reverse order, (RR). Then we commute it with C using (RC).

There is an important aspect to keeping the operations in a walk in the normal order. Assignments, as we know, define configurations of the Refal machine. In particular, if it is a walk-end, i.e. the assignment (*E* <- **e0**), then *E* is the whole of the configuration. A configuration is a set of ground expressions. A ground expression defines an exact state of the view-field of the Refal machine (or a part of the view-field), which determines the further development of the process in the view-field. Thus we should have good means to define configurations. A general Refal expression defines a set of ground expressions which includes all expressions that can be obtained by substitution of *any* admissible values for free variables. As we have seen, the actual configurations which appear in metacomputation very often obey some restrictions on their variables. Thus we can consider the combination RA as a single unit, a restricted general expression which defines a configuration that cannot be represented without using restrictions. The walk CRA can be seen as C(RA): a contraction followed by a restricted configuration.

In metacomputation, restrictions help to purge those branches which actually will never be used: this is their main reason for being. If, for instance, a certain configuration is restricted by (**# sX -> A**), and one of the branches in the development requires the contraction (**sX -> A**), we eliminate this branch. Restrictions make it possible to treat each branch in metacomputation separately. But in the final graph, where all branches are again together, the restrictions are not needed if we have been keeping the branches in the correct order; they will be taken care of automatically, through the sequential execution of branches. When we translate a Refal graph into a Refal program we skip restrictions; this assumes, of course, that we never changed the order or branches.

Consider a few examples of working with restrictions. Take this program:

```
F {A e1  =  B <F e1>;
   s2 e1  =  s2 <F e1>;
   B e1  =  C <F e1>;
    =  ;}
```

to be referred to as P_1, and let us complete the corresponding graph with restrictions:

```
(e0 -> <F e1>) {(e1 -> A e1) (B<F e1> <- e0)
                + (# e1 -> A e1)(e1 -> s2 e1) (s2 <F e1> <- e0)
                + (# e1 -> A e1)(# e1 -> s2 e1)(e1 -> B e1)
                          (C <F e1> <-e0)
                + (# e1 -> A e1)(# e1 -> s2 s1)(# e1 -> B e1)(e1 -> )
                          ( <- e0)
                }
```

Now commute **(# e1 -> A e1)** with **(e1 -> s2 e1)** in the second branch. In accordance with (RC), resolve the clash

s2 e1 : A e1 = (s2 -> A) (e1 <- e1)

Discarding the assignment, we come to the restriction **(# s2 -> A)**, the symbol variable **s2** in the second sentence cannot take the value **A**.

In the third branch we commute the RC pair

(# e1 -> s2 e1)(e1 -> B e1)

The clash is **B e1 : s2 e1**, which is resolved by the assignment **(B <- s2)**. There are no contractions, thus the list of contractions must be represented by the identical operation **I**. Since it is always successful, the corresponding restriction **(# I)** is the impossible operation **Z**. It is not necessary to do further commutations. The left side L_3 is a subset of L_2, so this sentence will never be used, and the corresponding branch in the graph must be eliminated.

For the last branch of the graph we compute

contr(*empty* : **B e1**) = **Z**

(here the list of *resolution terms* is empty, not the list of contractions in a resolution term, hence the matching is impossible). The restriction **(# Z)** is **I**, i.e. no restriction. This reflects the orthogonality of L_4 and L_3. The commutation of **(e1 ->)** with the other two restrictions also produces **I**. Finally, we have for P_1 the graph:

```
(e0 -> <F e1>) {(e1 -> A e1) (B <F e1> <- e0)
                + (e1 -> s2 e1)(# s2 -> A) (s2 <F e1> <- e0)
                + (e1 -> ) ( <- e0)
                }
```

It starts with the unrestricted configuration **<F e1>**. The configuration resulting from one step on the first branch is **B<F e1>**, again unrestricted. The walk-end configuration on the second branch is **s2 <F e1>**, where **s2** can take any value except **A**.

In this example the varlist at the node of branching was simply **(e1)**, so we operated on ordinary, not list, contractions (we wrote **e1 -> A e1**, instead of **(e1) -> (A e1)**, etc.) Now suppose the varlist at some node is **(e1)(s2)(s3)**, and the graph G_2 at the node is:

{ (e1 -> s2 e1 s2)(s3 -> A) ...
+ (e1 -> s2 s3) ...
+ (e1 -> s3 e1 s3)(s2 -> A) ...
}

where the walk-ends are left out. We want to complete the graph with restrictions. The first branch, of course, never includes restrictions. To compute the restrictions of the second branch, we first have to transform the involved contractions to the list form. The two individual contractions on the first branch must be applied sequentially to the varlist (the composition rule for contractions), which yields the list contraction:

(e1)(s2)(s3) -> (s2 e1 s2)(s2)(A)

and the list contraction of the second branch is

(e1)(s2)(s3) -> (s2 s3)(s2)(s3)

The clash to resolve is

(s2 s3)(s2)(s3) : (s2 e1 s2)(s2)(A)

As we know, different sets of variables never mix in the GMA and driving, so the same indexes can appear in different roles, and we see this in the clash above. We are interested in the contractions for the variables in the argument of the matching, i.e. those of the L_2 of the graph. The final assignments for the variables of the pattern, i.e. those of the L_1, are of no use for us. But before we come to the final assignments, we have to maintain some assignments for those variables as part of the algorithm. For the computer executing this algorithm the likeness of the variables on the different sides of the clash presents no problems. But for the human being it is confusing. Thus before executing the matching let us rename the variables of L_1 as **eX**, **sY**, **sZ**:

(s2 s3)(s2)(s3) : (sY eX sY)(sY)(A)

Now we go ahead with the matching. The first sub-clash is **s2 s3 : sY eX sY**. Projecting the left term, we have the assignment **s2 <- sY**. The sub-clash becomes **s3 : eX sY**, and we project the right-end term **sY** on **s3**. Since **sY** already has a value, we face the internal clash **s3 : s2**, which resolves as **s3 -> s2**. Now **eX** accepts the remaining part of the sub-argument, which is empty; the resolution of the first sub-clash is

(s3 -> s2)(s2 <- sY)(<- eX)

We substitute now the contraction part of it into the remaining part of the clash:

(s2)(s2) : (sY)(A)

where **sY** has already been assign the value **s2** (the contractions must also be executed over the values of the pattern variables, but in this case nothing changes, because none of them includes **s3**). Thus the second sub-clash, **s2 : sY** has the empty resoluiton. The third sub-clash, **s2 : A**, results in the contraction **s2 -> A**. The final result of the matching is

(s3 -> s2)(s2 -> A) (A <- sY)(<- eX)

Therefore, the restriction on the second branch is

(# (e1)(s2)(s3) -> (e1)(s2)(s3) / (s3 -> s2)(s2 -> A)) =
(# (e1)(s2)(s3) -> (e1)(A)(A))

We also can leave the graph in the restriction in the factorized form:

(# (s3 -> s2)(s2 -> A))

To find the restrictions on the third branch we compute two contraction sets:

contr((s3 e1 s3)(A)(s3) : (sY sZ)(sY)(sZ))
contr((s3 e1 s3)(A)(s3) : (sY eX sY)(sY)(A))

(where we again renamed the variables in the patterns). The reader can verify that both sets reduce to **(s3 -> A)**. Thus according to the general rule the restriction should be

(# (s3 -> A) + (s3 -> A))

but we can, of course, simplify it to **(# s3 -> A)**. The graph G_2 when complete with restrictions is:

{ (e1 -> s2 e1 s2)(s3 -> A) ...
+ (e1 -> s2 s3) (# (s3 -> s2)(s2 -> A)) ...
+ (e1 -> s3 e1 s3)(s2 -> A)(# s3 -> A) ...
}

As in the case of contractions, it should be born in mind that the interpretation of, and the operations on, restrictions depend on the full list of variables. When we factorize and write out individual contractions only, we must know the varlist in order to do correct transformations. Slightly modifying the example we gave in the context of contractions, consider the graph:

{ (e1 -> e1 s2) ...
+ (e1 -> A) ...
}

From this graph we cannot see whether **s2** is in the input varlist or not. But this is necessary to know in order to interpret the graph and find restrictions. If the varlist is **(e1)**, then **s2** is a new variable that can take any value, and the first branch completely screens the second. Formally, in order to find the restriction on the second branch we match:

$$\mathbf{(A) : (e1\ s2)\ =\ (\ <\text{-}\ e1)(A <\text{-}\ s2)}$$

and the resolution has no contractions. If, however the input varlist is **(e1)(s2)**, the match is:

$$\mathbf{(A)(s2) : (e1\ s2)(s2)\ =\ (s2 \text{->}\ A)(\ <\text{-}\ e1)(A <\text{-}\ s2)}$$

Now the restriction is **(# s2 -> A)**. The second branch of the graph will be used whenever **s2** is distinct from **A**.

3.6.3 DRIVING AS GRAPH NORMALIZATION

Given a contraction $C = (V \text{->}\ L)$, we shall call the assignment $(L <\text{-}\ V)$ *the conjugated assigment* of C, and denote it as $C^{\sim}$. The pair

$$CC^{\sim} = (V \text{->}\ L)(L <\text{-}\ V)$$

is in the normal order. It is the *specialization* of the varlist V by the contraction C. Indeed, we execute C, which transforms our varlist V into $\underline{\text{var}}(L)$, and then restore the old varlist with its old values by the assignment $C^{\sim}$. Thus the only effect of this pair is to check that the current values of V satisfy the contraction C, without actually changing any values. For every binding of V, the specialization $CC^{\sim}$ is equivalent either to **I**, or to **Z**. The same pair in the reverse order is always equivalent to **I**:

$$C^{\sim}C = (L <\text{-}\ V)(V \text{->}\ L) = \mathbf{I}$$

Using conjugated assignments we can write the following formula for the negation of a product of contractions:

$$(\#CC) \qquad (\#\ C_1C_2) = (\#\ C_1) + C_1(\#C_2)C_1^{\sim}$$

Its justification: two contractions applied sequentially say 'no' either if the first first one says 'no', or if the first says 'yes', but then the second says 'no'. The conjugated assignment $C_1^{\sim}$ is necessary to restore the original environment.

We derived above a commutation relation for the pair RC. There are two more pairs which violate the normal order: AC and AR. The AC pair, if we take into account the varlist coupling rules, is nothing else but the familiar clash:

$$(AC) \qquad (E <\text{-}\ V)(V \text{->}\ L) = E : L$$

Here a varlist takes on a value which is immediately required to contract. Together with the general formula for clash resolution,

$$E : L = \underline{\text{sum}}_k\ (\underline{\text{var}}(E) \text{->}\ L^k)(E^k <\text{-}\ \underline{\text{var}}(L))$$

this gives us a commutation relation which restores the normal order by putting all contractions before the assignments.

Consider the AR pair

$$(E \leftarrow V)(\# V \rightarrow L)$$

A varlist is bound to some values, after which a check is made whether the values comply with certain restrictions. It should be possible to find those restrictions on $\underline{\mathrm{var}}(E)$ which are sufficient and necessary for the assigned values of V not to contract to L. For instance, if we assign a symbol to **sX** and then check that it is not **A**, we also could start with checking that it is not **A**, and then make the assignment. The commutation relation is:

$$\mathbf{(s1 \leftarrow sX)(\# sX \rightarrow A)} \;=\; \mathbf{(\# s1 \rightarrow A)(s1 \leftarrow sX)}$$

The general formula is:

$$\text{(AR)} \qquad (E \leftarrow V)(\# V \rightarrow L) \;=\; (\# \underline{\mathrm{contr}}(E : L))(E \leftarrow V)$$

We exclude those values of the variables in E for which E will become contractible to L.

Adding the Refal graphs corresponding to all functions of a program, we form the total program graph G^{p}, which is a complete equivalent of the program:

$$\begin{aligned} G^{\mathrm{p}} = \quad & \{(\mathbf{e0} \rightarrow \langle F_1\ \mathbf{e1}\rangle)\, G(F_1) \\ & +(\mathbf{e0} \rightarrow \langle F_2\ \mathbf{e1}\rangle)\, G(F_2) \\ & \ldots \\ & \} \end{aligned}$$

where $G(F_i)$ is the graph for the function F_i, more precisely, for the configuration $\langle F_i\ \mathbf{e1}\rangle$. It is a sum of walks in the normal form with the input varlist (**e1**) and the output varlist (**e0**). One walk of G^{p} defines one step of the Refal machine over a primary active expression. Put $\langle E\rangle$ into the view-field by the assignment ($\langle E\rangle$ <- **e0**), and 'hit' it by G^{p}, i.e. concatenate G^{p} on the right:

$$(\langle E\rangle \leftarrow \mathbf{e0})\, G^{\mathrm{p}}$$

We have now a graph which is a program to evaluate $\langle E\rangle$. The actual computation or metacomputation according to this program will look as an equivalence transformation of this graph. Take first the simplest case when the active expression is a call of F_i with an object expression E_0 as its argument:

$$\begin{aligned} (\langle F_i\ E_0\rangle \leftarrow \mathbf{e0})\ & \{(\mathbf{e0} \rightarrow \langle F_1\ \mathbf{e1}\rangle)\, G(F_1) \\ & +(\mathbf{e0} \rightarrow \langle F_2\ \mathbf{e1}\rangle)\, G(F_2) \\ & \ldots \\ & \} \end{aligned}$$

Using the distributivity of addition and resolving the clashes for **e0**, we see that only the definition of F_i survives:

$$(E_0 \text{ <- } \mathbf{e1})\ G(F_i)$$

$G(F_i)$ is a sum of walks of the form

$$(\mathbf{e1} \text{ -> } L)\ (R \text{ <- } \mathbf{e0})$$

where L and R are the left and the right sides of a sentence. Thus we have the sum of walks

$$(E_0 \text{ <- } \mathbf{e1})\ (\mathbf{e1} \text{ -> } L)\ (R \text{ <- } \mathbf{e0})$$

The resolution of the clash $(E_0 \text{ <- } \mathbf{e1})(\mathbf{e1} \text{ -> } L)$ is the matching of the argument to the left side of a sentence. Since E_0 is an object expression, the result will be either **Z**, and the corresponding walks will be eliminated, or an assignment $(E \text{ <- } \underline{\text{var}}(L))$. The first of such walks,

$$(E \text{ <- } \underline{\text{var}}(L))\ (R \text{ <- } \mathbf{e0})$$

will be transformed by combining the two assignments into one assignment for **e0**, which gives the result of the step. All remaining walks in the graph can be discarded, because the first applicable sentence is always used. This was formulated in Sec. 3.4 as the Screening rule of function transformation, and we already used it in driving. For graphs it looks as follows: if there is a branch in a graph on which there are no contractions, all subsequent branches starting from the same node can be eliminated.

We have used the graph without restrictions, and the initial configuration without free variables. In the general case there will be free variables in the initial and other configurations, and restrictions in the graphs. The rules of equivalent transformation of Refal graphs will be used as a way to perform driving. We shall also allow restrictions on the variables in the initial configuration.

Let us formulate the general scheme of driving by graph transformation. Suppose we have a restricted configuration, i.e. a restriction-assignment pair R^cA^{c0}. Here we indicate by superscripts the input and output variable sets of operations (only one set for restrictions). The variables of the initial configuration are C-variables. Zero stands for the varlist (**e0**). The variables of the total program graph are P-variables. When we multiply the initial configuration by G^p, we have a sum of the walks of the form

$$R^cA^{c0}C^{0p}R^pA^{p0}$$

These walks are transformed according to the following scheme, where at each stage of transformation we enclose in parentheses the pair of operations for which an equivalence formula is used.

Walk	Replacement and the meaning of tranformation
1. $R^c(A^{c0}C^{0p})R^pA^{p0}$	$C^{cc'}A^{c'p}$

		Matching argument to the left side. C'-variables are derived from C-variables.
2.	$(R^{c}C^{cc'})A^{c'p}R^{p}A^{p0}$	$C^{cc'}R^{c'}$ Restrictions on the initial variables are translated into restrictions on the derived variables.
3.	$C^{cc'}R^{c'}(A^{c'p}R^{p})A^{p0}$	$R^{c'}A^{c'p}$ Restrictions on P-variables are translated into restrictions on C'-variables.
4.	$C^{cc'}(R^{c'}R^{c'})A^{c'p}A^{p0}$	$R^{c'}$ Composition of restrictions.
5.	$C^{cc'}R^{c'}(A^{c'p}A^{p0})$	$A^{c'0}$ Composition of assignments.
6.	$C^{cc'}R^{c'}A^{c'0}$	Normal form.

Making such transformations along every branch in the program graph G^{p}, we come to a graph where all walks are in the normal form. What we achieved is, essentially, a transformation of a program requiring two steps of the Refal machine into a program which does the same thing in one step. The pair RA in the original walk requires a check on the input variables according to R, and a substitution according to A. This can be seen as a step of the Refal machine described by the normal walk CRA with the trivial contraction C = **I**. Thus we transformed IRACRA into CRA. If we have a graph in the normal form (i.e. a sum of normal-from walks, possibly factorized by the use of the distributive law), we can 'hit' every walk-end which assigns a primary active expression to **e0** by G^{p}, which will give us a graph where some of the walks describe two steps of the Refal machine. Then we transform these walks into the normal form, which requires only one step more than in the above transformation:

1. CR(AC)RA
2. C(RC)ARA
3. (CC)RARA
4. CR(AR)A
5. C(RR)AA
6. CR(AA)
7. CRA

The extra step is from stage 3 to stage 4, the composition of contractions; we could avoid this step by leaving the contractions factorized: CCRA. Now we have a program which is executed in one step of the Refal machine.

The process of transformation of a Refal graph to the normal form will be referred to as *normalization*. Hitting a primary active walk-end (**<*E*> <- e0**) by the total program graph G^{p} with the subsequent normalization is a form of driving. Its advantage for further analysis and computeriza-

tion is that the process is broken down into simple elementary steps, and at each step we have a meaningful representation of the current situation -- a Refal graph.

3.6.4 DEFINING AND DELETING VARIABLES

As we know, not every graph is executable. In outsde-in driving we have sometimes to return to a previous point and make a different decomposition of an active expression. Decomposition is an equivalent transformation of a graph -- the formula for composition of assignments used in the reverse order. We are free to decompose active expressions in any way. When we meet a hindrance, we extract it by decomposition and make of it a separate subgraph, without redoing any part of the work already done. If

$$(\langle Q \rangle \text{ <- e0})\ G^{\mathrm{P}}$$

is unexecutable, and Q_1 is the hindrance in Q, we decompose:

$$(\langle Q \rangle \text{ <- e0}) \ = \ (\langle Q_1 \rangle \text{ <- eX})(\langle Q_2 \rangle \text{ <- e0})$$

and move G^{P} inside, forming the subgraph assignment:

$$(\{(\langle Q_1 \rangle \text{ <- e0})\ G^{\mathrm{P}}\} \text{ <- eX})\ (\langle Q_2 \rangle \text{ <- e0})$$

We are also free to hit by G^{P} and normalize (i.e. to drive) any primary walk-end in the main graph or in any subgraph. In normalization we are free to use composition and commutation relations at any point of the graph.

We need a few more equivalence transformations concerning subgraphs. The following distributive relation is obviously valid:

$$(\{G_1 + G_2\} \text{ <- e}I)G_3 \ = (\{G_1\} \text{ <- e}I)G_3 + (\{G_2\} \text{ <- e}I)G_3$$

When a subgraph is reduced to a walk, we can take out some, or all, of its elements. Consider such a subgraph in its context:

$$W_1\ (\{W\} \text{ <- e}I)\ G_2$$

If the walk W starts with a contraction C, the whole walk is applicable only if C is successful. Thus we can take C out of the subgraph and put it right after W_1. This, however, will ruin the after-subgraph part G_2 if we do not restore the environment in which G_2 is supposed to operate, namely, the one that was before the contraction C. This restoration, as we know, is achieved by the conjugated assignment $C^{\sim}$. Thus we have the formula:

$$W_1\ (\{C\,W\} \text{ <- e}I)\ G_2 \ = \ W_1 C\ (\{W\} \text{ <- e}I)\ C^{\sim}\ G_2$$

For a walk starting with a restriction it is not necessary to restore the environment, since it does not change:

$$W_1\ (\{R\,W\} \text{ <- e}I)\ G_2 \ = \ W_1 R\ (\{W\} \text{ <- e}I)\ G_2$$

For a walk which consists only of a walk-end, we reassign to the subgraph variable **e***I* the value assigned to **e0**. It is here that we need the operation def in order to add **e***I* to the varlist and assign to it the computed value:

$$W_1\ (\{(E \leftarrow \mathbf{e0})\} \leftarrow \mathbf{e}I)\ G_2 \ = \ W_1\ \underline{\mathrm{def}}(E \leftarrow \mathbf{e}I)\ G_2$$

The delete operation del is complementary to def. Variables can be deleted from the current varlist at a point before an assignment ($E \leftarrow V$) if var(E) is a proper subset of the output varlist of the preceding walk. For instance, in the walk

((e4)(s5)(s7) <- (s5 e4 s7)(s5)(s7)) (<F2 e4 s7> <- e0)

we can insert a delete operation

((e4)(s5)(s7) <- (s5 e4 s7)(s5)(s7)) del(s5) (<F2 e4 s7> <- e0)

which may be useful when we implement Refal graphs as computer programs, but is redundant, because we can always establish that a variable is deleted at a certain point in a graph by comparing the output varlist before that point with the input varlist after it.

When variables are deleted, we may be able to make some improvements in the graph. Consider a walk of the form

$$C\ R\ \underline{\mathrm{del}}(v)\ (E \leftarrow \mathbf{e0})$$

where C is some contraction, R a restriction reflecting the position of the walk in a graph, and the variable v is not used in E. Then the value of v disappears, and will never be used again in metacomputation. Indeed, when we hit the walk-end by G^0 and resolve the clash for **e0**, only the variables in E matter. The restriction R may include some requirements to v, and after v disappears, these requirements become irrelevant; we should take that into account and simplify R into some R'.

This transformation can be written as the rule:

$$R\ \underline{\mathrm{del}}(v) \ = \ R\ \underline{\mathrm{del}}(v)\ R'$$

We left R where it was, because we still want to check the full restrictions to decide on the applicability of the walk. But for future development of the walk by driving we will use R' as the restriction on the configuration E. As we have already mentioned, the combination RA in the normal form of a walk stands for a restricted configuration. We use restrictions in order to avoid unfeasible walks in driving. After v is deleted, it is R' that will be used for that purpose.

Let us now figure out how to transform R into R'. The walks of the form (V -> L) in the contraction graph negated by R can be, obviously, deleted. But this may not be enough. It may happen that v enters restrictions for other, not deleted, variables. Since e-variables are never repeated in contractions, this may happen only if v is an s-variable. Take this program:

```
F1 { (s5 e4 s7)(s5)(s7)  =  <F2 e4 s7>;
     (e4)(s5)(s7)  =  <F3 e4 s7>;
   }
```

The first walk of the graph G(**F1**) for this program we have used as an example above. The second walk is:

(e1 -> (e4)(s5)(s7)) (# e4 -> s5 e4 s7) (< F3 e4 s7 > <- e0)

The restriction here, which we take as R, does not allow **e4** to start and end with symbols identical to the values of **s5** and **s7**, respectively. Then the variable **s5** disappears. What is the restriction R' on the remaining variables of the walk-end?

The reader can figure out that in this situation there must be no restrictions on **e4** at all, i.e. R' should be empty (identical operation **I**). We are going to prove this by proving that for any value of **e4** and **s5** we can a value of the deleted variable **s5**, such that R will let pass the binding. This will mean that the walk-end configuration is unrestricted.

So let us see what values of **e4** may be stopped by R. Obviously **e4** must end with **s7**, otherwise R will definitely not stop it. By the same token, **e4** must start with some symbol S. So, **e4** $= S\,E$ **s7**, where E is some expression. Take any symbol S' distinct from S and assign it to **s5**, i.e. form the binding:

$$(S\,E\ \mathbf{s7})(S')(\mathbf{s7}) \ \mathbf{<-}\ \ (\mathbf{e4})(\mathbf{s5})(\mathbf{s7})$$

When R is executed with this binding, it lets it pass. This proves the point.

Generalizing this argument, we come to the following theorem.

Theorem. Let

$$R \ = \ (\#\,(V \text{-> } L_1) + (V \text{-> } L_2) + \ldots + (V \text{-> } L_q))$$

be the restriction on the varlist V just before an assignment (E <- **e0**), and let $V = V^n V^d$, where $V^n = \underline{\text{var}}(E)$, so that V^d is the list of deleted variables. Then the restriction R' which describes the configuration E can be formed from R as follows. First, remove the sublist V^d from V and the corresponding sublist in L_i for every negated term in R, thus transforming it into $(V^n \text{-> } L_i^n)$. Then delete every term where there is at least one occurrence of a variables from V^d in L_i^n.

Proof. We want to find out what restrictions are put on V^n because of the restriction $(\#\ V \text{-> } L_i)$ on V. To exclude some value-list L^n from V^n, one must be sure that $L^n L^d$ is removed by R from $V^n V^d$ for every possible value-list L^d of V^d. If there are no occurrences of V^d-variables in L_i^n, then any value list of object expressions in $L_i{}^n$, combined with any value-list of object expressions from $L_i{}^d$, will become a value-list to be excluded form V. Therefore, $(\#\ V^n \text{-> } L^n)$ is the correct restriction on V^n. If, however, L_i^n includes an occurrence of sI from V^d, then none of the object expressions of L_i^n can be excluded. Indeed, when we pick up an object expression from the class represented by the pattern L_i^n, we give an object value to every free variable in it. Suppose the value given to sI is S, and the resulting object expression (value-list) is L_i^{no}. If we also put sI -> S while contracting L_i^d to L_i^{do}, then the value-list $L_i^{no} L_i^{do}$ for V is an instance of $L_i^n L_i^d$, and will be removed by R. But we can give to sI a different value S'. The object expression thus obtained

obviously cannot be recognized as $L_i^n L_i^d$, and R will not remove it. Hence the criterion is not satisfied, and no object expression from L_i^n can be excluded. The corresponding restriction term must be deleted.

THE SUMMARY OF GRAPH TRANSFORMAITONS

In the following E stands for an expression, L an L-expression, V a varlist, W a walk, G a graph. If $C = (V \text{->} L)$, then $C^{\sim} = (L \text{<-} V)$.

1. The formats of contractions (C), restrictions (R), assignments (A), and subgraphs (S):

(C) $[V]\,(V \text{->} L)\,[\underline{\text{var}}(L)]$

(R) $[V]\,(\#\,(V \text{->} L_1) + \ldots + (V \text{->} L_q))\,[V]$

(A) $[V]\,(E \text{<-} V')\,[V']$ where $\underline{\text{var}}(E) \leq V$

(S) $[V]\,(\{[V_g]G\} \text{<-} \mathbf{e}I)\,[V(\mathbf{e}I)]$ where $V_g \leq V$

2. Varlist coupling in composition and addition

$$[V]\,W_1\,[V']\,G_2\,[V''] = [V]\,W_1\,G_2\,[V'']$$
$$[V]\,G_1 + [V]\,G_2 = [V]\,\{G_1 + G_2\}$$

3. Identity operations

$$\mathbf{I}\,W = W\,\mathbf{I} = W$$
$$\mathbf{Z} + G = G + \mathbf{Z} = G$$
$$(\#\,\mathbf{I}) = \mathbf{Z}$$
$$(\#\,\mathbf{Z}) = \mathbf{I}$$

4. Distributive laws:

(D.L) $W(G_1 + G_2) = WG_1 + WG_2$

(D.R) $(\{G_1 + G_2\} \text{<-} \mathbf{e}I)G_3 = (\{G_1\} \text{<-} \mathbf{e}I)G_3 + (\{G_2\} \text{<-} \mathbf{e}I)G_3$

5. Composition

(CC) $(V_1 \text{->} L_1)\,(V_2 \text{->} L_2) = (V_1 \text{->} L_1/(V_2 \text{->} L_2))$

(AA) $(E_2 \text{<-} V_2)\,(E_1 \text{<-} E_1) = ((E_2 \text{<-} V_2)/E_1 \text{<-} V_1)$

(RR) $(\#\,G_1)(\#\,G_2) = (\#\,G_2)(\#\,G_1) = (\#\,G_1 + G_2)$

(#CC) $(\#\,C_1C_2) = (\#C_1) + C_1(\#C_2)C_1^{\sim}$

6. Commutation:

(AC) $(E \text{<-} V)(V \text{->} L) = E : L = \underline{\text{sum}}_i(\underline{\text{var}}(E) \text{->} L^i)(E^i \text{<-} \underline{\text{var}}(L))$

(RC) $(\#\,V \text{->} L_1)(V \text{->} L_2) = (V \text{->} L_2)(\#\,\underline{\text{contr}}(L_2 : L_1))$

(AR) $(E \text{<-} V)(\#\,V \text{->} L) = (\#\,\underline{\text{contr}}(E : L))(E \text{<-} V)$

7. Exiting a subgraph:

$$W_1\,(\{C\,W\} \text{<-} \mathbf{e}I)\,G_2 = W_1 C\,(\{W\} \text{<-} \mathbf{e}I)\,C^{\sim}\,G_2$$
$$W_1\,(\{R\,W\} \text{<-} \mathbf{e}I)\,G_2 = W_1 R\,(\{W\} \text{<-} \mathbf{e}I)\,G_2$$
$$W_1\,(\{(E \text{<-} \mathbf{e}0)\} \text{<-} \mathbf{e}I)\,G_2 = W_1\,\underline{\text{def}}(E \text{<-} \mathbf{e}I)\,G_2$$

3.7 Elementary contractions

When we factorize an individual contraction *v* -> *L*, we represent it as a composition of two non-trivial contractions. This process can be repeated until it comes to its natural end when *L* is such that the contraction cannot be broken down in this way. Such a contraction will be called *elementary*. In the algorithm of generalized mapping we move on with minimal steps when we impose contractions on the variables in the argument; these contractions are exactly the elementary contractions of Refal. There are seven of them:

e*I* -> s*J'* e*I*	**S** *I J'*
e*I* -> (e*J'*)e*I*	**B** *I J'*
e*I* -> e*I* s*J'*	**T** *I J'*
e*I* -> e*I*(e*J'*)	**C** *I J'*
e*I* -> *empty*	**X** *I* **N**
s*I* -> *S*	**I** *I S*
s**I** -> s*J*	**O** *I J*

As in the GMA, the variables with the index *J'* are *new* variables. i.e. such that have not yet been used in the preceding factors of the overall contraction. The first elementary contraction in this list must be understood as the requirement that e*I* starts with *any* symbol, and the next three contractions are to be read in the similar way. On the contrary, the last contraction uses the variable s*J* (not s*J'*), which is *old*, i.e. is assumed to have a definite value in the execution of the walk, for which it must have been already used in the right side of a contraction or assignment. The requirement is that the value of s*I* is distinct from that of s*J*.

In programming, elementary contractions can be represented in a compact code. We use in the supercompiler the three-symbol code which is shown in the second column above. The first symbol codes the type of the contraction. The mnemonics is: S -- Symbol variable; B -- Bracket; T -- the following letter of the alphabet after S, to denote a symbol variable, but on the other end; C -- the same for Bracket; X -- cross out the expression; I -- the symbol variable Is...; O -- Old variable. The second symbol is always the index of the variable being contracted. The third symbol is either the index of another variable, or a symbol used in the contraction (**N** in the X-contraction is for the even count, so as to get every elementary contraction by the pattern **s1 s2 s3** when programming in Refal.

There are two more contractions which we often treat as elementary, even though they can be factorized:

e*I* -> *S* e*I*	**L** *I S*
e*I* -> e*I S*	**R** *I S*

(Mnemonics: symbol on the Left and Right.)

Some care is needed in dealing with elementary contractions of the **O** type, s*I* -> s*J*. Our notation has the simplified form of an individual contraction, but it actually cannot be expressed by an individual contraction. As we stressed before, s*J* is an *old* variable, which enters the current varlist. The exact definition of the **O**-type contraction is:

$$(sI \text{ -> } sJ) \quad = \quad (sI)(sJ) : (sJ)(sJ)$$

The matching checks that the values of *sI* and *sJ* are equal, and tells us to use *sJ* as the variable for this common value. When we use the individual contraction **sI -> sJ** for substitution, it is completely equivalent to the full list form. But when we trace a contraction chain to determine where and how the variables get defined, the simplified individual form may mislead. You need to observe the full form in order to see that *sJ* gets redefined as equal to what it was before. The simplified form creates an impression that *sJ* takes over the value of *sI*. But this would be true only for the symbol *assignment*, which is defined as

$$(sI \text{ <- } sJ) \quad = \quad (sJ) : (sJ)$$

Thus the general rule that the arrows we use in contractions and assignments can be simply replaced by the matching operation sign must be qualified by the definition of the O-type contraction.

Given a contraction **e1 -> *L***, we can factorize it into elementary contractions by matching **e1** : *L*. The sequence of contractions resulting from the match (which, of course, always succeeds) is equivalent to the original contractions, but will be expressed in terms of **e1** and its derivatives. The assignments resulting from the matching express the variables of *L*) in terms of those new variables. Example: if

$$L \quad = \quad \mathbf{(sX)s5\ eY\ s5}$$

the sequence of elementary contractions resulting from the matching **e1** : *L* is:

(e1 -> (e2)e1) (e2 -> s3 e2) (e2 ->) (e1 -> s4 e1)
(e1 -> e1 s5) (s5 -> s4)

In the code described above this is

B12 S23 X2N S14 T15 O54

Folding these contraction into one, we get

e1 -> (s3)s4 e1 s4

If we want a factorization which retains the old output variables, we can use the assignment part of the resolution, which is

(s3 <- sX) (e1 <- eY) (s4 <- s5)

in order to make substitutions in the direction opposite to the arrows.

This procedure or restoring the old output variables looks unnatural and is rather messy. But we do not actually need it for metacomputation. When we want to decompose the left side of a sentence **< F *L* >** = *R* into elementary contractions, we match **e1** : *L*, take the sequence of contractions as the new left side, and substitute the assignments into *R* to produce the new right side. This will be the original sentence in the graph form. We can bring it back to

the program form by folding the chain of contractions into one contraction. Now the whole sentence is expressed in terms of derivatives of **e1**, instead of the original variables, but this is not bad at all. On the contrary, this can be used for standartization of Refal sentences and graphs with respect to the naming of variables. If we accept a certain order of projecting in the GMA, and a certain way to produce new variables (e.g. the smallest number not yet in use), then the sentences that differ only by the names of variables will be reduced to exactly the same normal form.

The use of elementary contractions in the coded form makes it possible to atomize a great deal of the objects of transformations when we work with Refal graphs, and develop efficient algorithms of algebraic manipulation of these objects.

3.8 The strategy of metacomputation

Driving is not the only party to metacomputation. If we simply drive on every active configuration as it appears in the graph, we will never stop in most cases. The *strategy* of metacomputation is a method to curb driving and arrive at a finite graph of states. To do that we have sometimes to loop back from a newly created configuration to one of the already existing in the graph. Those configurations which we are prepared to leave in the final graph will be called *basic*. It is possible to loop back from a configuration Q to a basic configuration B if Q can be reduced to B, i.e. represented as its subset:

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

Then instead of calling (putting into the view-field) Q, we execute the reduction assignments and call B:

$$(Q \leftarrow \mathbf{e0}) = (E \leftarrow \underline{\mathrm{var}}(B))\,(B \leftarrow \mathbf{e0})$$

Let us consider some cases of configurations we create in driving and relations between them (see Fig. 3.2).

The configuration Q_2 gives rise to two passive configurations (we represent such by rectangles). This is a natural end of driving. We do not have to worry about Q_2 any more.

Observing configuration Q_4 and comparing it with its predecessors Q_3 and Q_1, we may decide that none of the predecessors is close enough to Q_4 to try looping back. Then we go on driving.

Configuration Q_5 is a subset of Q_3; in particular, the two may be equal. This is the case when we certainly must loop back. The reduction arc is shown in the Figure by a broken line.

Configuration Q_6 goes over into Q_7 no matter what the values of its variables. The arc from Q_6 to Q_7 may or may not carry a contraction. We shall call such configurations as Q_6 *transitory*. There are good reasons to purge transitory configurations from the graph as they appear. The arc from Q_4 must then be redirected straight to Q_7, and the possible contractions on the Q_6-to-Q_7 arc must be added. In this way we avoid a great deal of the metacomputation overheads in a situation where metacomputation is, in fact, computation. The disadvantage of this strategy is that if the initial program allows infinite computation processes for some input data, the process of metacomputation may aslo become infinite. If one of the goals of metacomputation is to find infinite loops in the program (i.e. *some* of them), then transitory configurations should not be automatically skipped.

The relation between the configurations Q_8 and Q_9 is the hardest case, when, on the one hand, Q_9 is not a subset of Q_8, but on the other hand, it is, in some sense, too close to Q_8 to simply go on driving. We foresee that if we drive on we can be constructing an infinite row of close configurations. In this case we *generalize* Q_9 and Q_8, i.e. find a configuration Q_{10}, such that both Q_9 and Q_8 can be reduced to it. Then we delete the whole development of Q_8, reduce Q_8 to Q_{10}, and develop Q_{10}.

Consider the following example of the last case. The program is:

```
Fab {eX = <Fab1 ()eX>;}
Fab1 {
  (eY)A eX = <Fab1 (eY B)eX>;
  (eY)sZ eX = <Fab1 (eY sZ)eX>;
  (eY) = eY;}
```

and the configuration to meta-evaluate is **<Fab e1>** (we want simply to reproduce the program).

The total program graph G^P is:

```
(G^P)   { (e0 -> <Fab eX>) (<Fab1 ()eX> <- e0)
        + (e0 -> <Fab1 eX>)
            { (eX -> (eY)A eX) (<Fab1 (eY B)eX> -> e0)
            + (eX -> (eY)sZ eX) (<Fab1 (eY sZ)eX> -> e0)
            + (eX -> (eY)) (eY <- e0)
            }
        }
```

We use letters for the indexes of P-variables, in order to readily distinguish them from the numbered C-variables (in porgrams both sets have numbers as indexes). The initial graph is

(1) $(Q_1$ <- e0)
(Q_1) <Fab e1>

To make one step of driving, we hit it by G^P and normalize using the procedure already familiar to the reader. The result is

(2) $(Q_2$ <- e0)
(Q_2) <Fab1 ()e1>

The construction of the graph of states, however, is not simply driving. We must remember the history of computation, the past states of the computing system. We shall put **P** in front of past configurations, written, as always, in the form of assignments. The graph of states at this point should have looked as:

(3) P(Q_1 <- e0) (Q_2 <- e0)

but the initial node (1) is transitory, so we do not keep it, and (2) is the right graph, after all.

Now we develop Q_2. Consider this in greater detail. We first reserve Q_2 as a past configuration, then make another copy and hit it by the program graph:

P(Q_2 <- e0) (Q_2 <- e0) G^P

After the resolution of the clash for **e0**, we have:

```
P(Q2 <- e0) (()e1 <- eX)
        { (eX -> (eY)A eX) (<Fab1 (eY B)eX> <- e0)
```

+(eX -> (eY)sZ eX) (< Fab1 (eY sZ)eX > <- e0)
+(eX -> (eY)) (eY <- e0)
}

Now we want to normalize the first branch. Formally, we use the distributive law for graphs. The assignment for **eX** is duplicated and left before the remaining branches:

P(Q_2 <- e0) {(()e1 <- eX) (eX -> (eY)A eX) (< Fab1 (eY B)eX > <- e0)
+(()e1 <- eX){(eX -> (eY)sZ eX) (< Fab1 (eY sZ)eX > <- e0)
+(eX -> (eY)) (eY <- e0)
}
}

Normalization of the first branch results in

P(Q_2 <- e0) { (e1 -> A e1) (Q_3 <- e0)
+ ...
}

(Q_3) **(< Fab1 (B)e1 >)**

Before developing the remaining branches, we go on with the transformation of the node on the first branch; this method is known as the *depth-first* construction of a tree. Consider Q_3, and compare it with its predecessor Q_2. None is a subclass of the other. But they are 'dangerously close'. If we further drive Q_3, we shall have at the first branch from each node:

< Fab1 (BB)e1 >
< Fab1 (BBB)e1 >
...

etc., infinitely. We have, therefore to generalize Q_3 and Q_2. We shall discuss the possible algorithms of generalization in Chapter 6. At this point, we simply replace the subexpression that causes the difference by a free variable, to come to the generalization:

(Q_4) **< Fab1 (e2)e1 >**

Now we eliminate the past configuration Q_2 together with its development, reduce Q_2 to Q_4, and develop Q_4:

(4) **(<- e2) P(Q_4 <- e0) (Q_4 <- e0)** G^p

While it should be clear that this step of transformation is perfectly justified, the formal aspects requires some discussion. When a generalization takes place, we can consider all variables V' of the generalized configuration as *new*, and make them all textually different from the current C-variables V. By matching the previous configuration to the generalized one, we find the reduction assignments (E <- V'), where $\underline{\text{var}}(E) = V$. Since no variable from V' enters E, the reduction assignments can be factorized into a sequence of individual assignments in any order. Now consider those factors which have

the form (v <- v'). For each such factor we can rename the variables v' into v, i.e. restore its old name, and then delete the factor. This can be done because, again, no variable from V' is used in E. After the reduction assignment we can delete, using del, those variables form V which do not enter the modified V'. In our graph **e1** survives generalization, and a new variable, **e2**, appears in the varlist.

Normalizing the first branch in (4),

(<- e2) P(Q_4 <- e0) {(e1 -> A e1) (<Fab1 (e2 B)e1> <- e0)
+ ...
}

we have a configuration which can be reduced to Q_4. To find the reduction assignments, we match

<Fab1 (e2 B)e1> : <Fab1 (e2)e1>

Even though it is clear that the solution in this case can be given by the individual assignment (**e2 B <- e2**), the optimal form of the reduction to a former configuration generally cannot be immediately obtained from matching. Unlike the case of the reduction to a generalization, the variables of the configuration to which the reduction takes place are not new, and therefore cannot be given new indexes. In the matching

$$Q^{\mathsf{after}} : Q^{\mathsf{before}}$$

the variables of Q^{after} are derived from those of Q^{before}. To see how this may interfere with factorization of assignment, let us imagine that we want to reduce **<F (e2)e1>** to **<F (e1)e2>**. The matching results in the list assignment

(e1)(e2) <- (e2)(e1)

which is, of course, a correct solution, and stands for a simultaneous substitution. If, however, we tried to see it as a pair of individual assignments:

(e1 <- e2) (e2 <- e1)

this, obviously, would be an error.

It is worthwhile to make a digression on the factorization of list assignments. To reduce the list assignment above to a sequence of individual assignments, we must introduce an intermediate configuration, co-extensional with Q^{before}, but with a disjoint set of variables. We do that in order to be able to factorize the list assignments involved. We have now two matching operations:

(<F (e2)e1> : <F (eY)eX>) (<F (eY)eX> : <F (e1)e2>)

Both can be factorized with the result:

(e2 <- eY) (e1 <- eX) (eY <- e1) (eX <- e2)

This solution can be improved. We can formulate the following commutation-rule for individual factor-assignments:

$(E_1 \leftarrow v_1)(E_2 \leftarrow v_2) = ((E_1 \leftarrow v_1)/E_2 \leftarrow v_2)(E_1 \leftarrow v_1)$
if v_2 is not in $\underline{\text{var}}(E_1)$

Exercise ... Prove this rule.

We commute the third and the fourth assignments, then the second and the third:

(e2 <- eY) (e1 <- e2) (e1 <- eX) (eY <- e1)

Now the assignment for **eX** can be deleted because it is not used later, and **eX** is not in the output varlist **(e1)(e2)**:

(e2 <- eY) (e1 <- e2) (eY <- e1)

This is the efficient solution. No further transposition or deletions can be made.

Exercise. Show formally how to find the individual assignment sequence for the reduction:

< F s1 s2 s3 > : < F s2 s3 s1 >

Returning to our graph, we see that on the second branch we also have the reduction case. We finally get the graph:

(<- e2) P(Q_4 <- e0) {(e1 -> A e1) (e2 B <- e2)(Q_4 <- e0)
+ (e1 -> s3 e2) (e2 s3 <- e2)(Q_4 <- e0)
+ (e1 ->) (e2 <- e0)
}

which is an exact equivalent of the original program.

Note that because of the depth-first order of graph construction we avoided the development of all branches except the first one at the node Q_2. If we used the breadth-first approach, we would have developed all branches, and then discovered that it was unnecessary, since the whole development disappears and the graph is redrawn for the generalized configuration Q_4. With the depth-first method, we have come to the generalized configuration on the basis of the first branch only, and this configuration was good enough to secure the looping-back on the other branch. This situation is typical and speaks strongly in favor of the depth-first graph construction, even though it is easy to set up a situation where each new branch requires a new generalization.

Exercise... Give an example where such a situation takes place.

There is a great deal of freedom in deciding how to go on with metacomputation. The strategy of metacomputaiton defines which of the possible actions must be taken at each stage of the process. The list of actions that may be possible includes these:

1. Pick up one of the configurations in the graph and reduce it to another configuration. We have the option of reducing only to a direct ancestor, or to any configuration in the already constructed part of the graph.
2. Pick up an end-configuration, decompose it, and and drive the primary active subexpression. It still remains to decide what method to use in choosing the configuration, e.g., depth-first or breadth-first, and how to decompose, e.g., outside-in or inside-out.
3. Declare a configuration *basic*. Then it will neither be reduced to another, nor driven further. The list of basic configurations could be given at the beginning, or a more sophisticated decision procedure could be used.
4. Generalize a configuration with one of the preceding configurations Q^{before}, with the following reduction of Q^{before} to the generalization. The operation of generalization still is to be defined in some way.
5. Stop and output the graph. For this to be admissible, each node in the graph must be developed or reduced to another node, and there must be no loops consiasting of reduction arcs only. In this way the strategy guarantees that the graph can serve as the program for the initial configuration.

The crucial point in elaborating a strategy of metacomputation is to ensure that the process of graph construction is finite. One way to do it is to define, in some way, the full set B of basic configurations, and to prove that (1) B is finite, (2) for every ground configuration Q which can appear in the view-field under a given program, there is such a configuration in B to which Q is reducible. The strategy of metacomputation must then include the provision that each primary active configuration is reduced to one of the basic configurations. Two variations are possible. First, we can always drive on transitory configurations, with the consequence that the finiteness will be guaranteed only if the program graph G^{p} guarantees it. Second, we can try to reduce each new configuration to one of its predecessors (which will be called a *recursive* configuration in case of success) and only in case of a failure reduce it to one of the predefined basics from B. Then the total set of basic configurations will include, in addition to the predefined basics, those recursive configurations which were found during the graph construction.

Another approach to ensuring the finiteness of metacomputation is to produce basic configurations by generalizaiton on the fly, but limit to such a family of potential basic configurations that a sequence of possible generalizations will always be finite. If we set that all basic configurations must have the form $<F\ L>$, where L is an L-expression, then the following theorem warrants the finitness of metacomputation:

Theorem. A chain of L-patterns $L_1 < L_2 <$ ... etc., where each pattern is a proper subset of the next pattern, can only be finite.

Exercise ... Prove this theorem. Prove also that for restricted patterns this theorem does not hold.

The concept of the set of basic configurations makes it possible to quantify the interpretation-compilation dimension of programs well-known in programming. We call a program interpretive if it tends to determine which action to take by analyzing some data. Such a program can be shifted toward the compilation end of the interpretation-compilation axis by analyzing some of those data and including the necessary actions right in the program, *compiling* the instructions necessary to do the job. The more compilative program will work faster than the more interpretive one, because it will not spend time for analyzing some part of the data. On the other hand, it may be considerably greater than the interpretive program at the expense of the detailed instructions. We can regulate the position of the metacomputed program on the interpretation-compilation axis by varying the set of basic configurations. The larger (more extensive) the basic configurations are, the more interpretive will be the program. When we partition basic configurations into more detailed configurations, the program becomes more compilative. Correspondingly, we can speak of more interpretive and more compilative strategies of metacomputation. Programs developed under varying strategies remain, of course, equivalent. By mechanizing metacomputation we can have a mechanized way of controlling the interpretation-compilation feature according to our needs.

We shall now formulate two generally applicable, and very interpretive, strategies of metacomputation.

The first strategy is *generalization to functions*. The set of basic configurations consists of all configurations of the form $<F\ \mathbf{e1}>$, where F is a function defined in the program. Each active configuration is decomposed according to the inside-out principle, and the primary active subexpression is reduced to a basic configuration. It is easy to see that this strategy reproduces the Refal program; we saw an example above (function **Fab**).

The second strategy is *generalization to formats*. The basic configurations have the form $<F\ L>$, where L is a pattern of the argument of F, an L-expression. Each active configuration is, again, decomposed using the inside-out principle, but L for each function is not given in advance; it is determined in the process of metacomputation. We take the generalization of all left sides of the function definition as the tentative basic configuration. When a call of F is met in driving, it is generalized, if necessary, with the tentative basic, and the generalization is taken as the new tentative basic. We do not discuss generalization here, because it will be discussed later in the book; a simple technique, analogous to the *unification* algorithm used in term-rewriting systems, will usually produce good results.

To get an idea of the effect a strategy has on the resulting program, take this simple initial program:

```
F {
   (e1)(A e2)(e3)  =  <F (e1 B)(e2)(e3)>;
   (e1)()(e3)  =  e1 e3; }
```

and compare the results of metcomputation with the two strategies we have defined. With the generalization to functions the Refal graph is:

```
(e0 -> Q1) Q1{(e1 -> (e2)(A e3)(e4)) ((e2 B)(e3)(e4) <- e1) Q1
             +(e1 -> (e2)()(e4)) (e2 e4 <- e0)
             }
```

(Q_1) **< F e1 >**

With the generalization to formats it is:

```
(e0 -> Q2) Q2{(e2 -> A e2) (e2 B <- e2) Q2
              + (e2 -> ) (e1 e3 <- e0)
             }
```

(Q_2) **< F (e1)(e2)(e3) >**

Since Q_2 is a subset of Q_1, the first graph is on the interpretive, while the second on the compilative, side. The second graph treats **F** as a function of three arguments, in agreement with the intuitive meaning of the definition. When one argument is checked or updated, there is no need to deal with the others. The first graph considers **F** as a function of one variable **e1**, which is interpreted as a list of three subexpressions. At each step the three subexpressions are extracted from the value of **e1**, then each is treated as necessary, and then they are assembled again, using parentheses, into a value for **e1**. Clearly, this is much less efficient procedure.

The straightforward Refal interpreter treats Refal functions as functions of one variable, as in the first graph above. Hence a substantial effort may go into assembling and disassembling of the argument in cases where different parts of it could be treated as distinct arguments. By transforming Refal programs into graphs using generalization to formats, and implementing such graphs, a more efficient implementation of Refal can be developed.

In our example, all the advantages have been on the side of the compilative version, because Q_2 contracts Q_1, and no new configurations become necessary. We simply correct the excessively inflated basic configuration. Generally, however, making the basic configurations more precise increases their number. Then the graph becomes larger.

3.9 Metacode

To write Refal programs that deal with Refal programs (or graphs), we have to represent Refal programs as *object* expressions. Indeed, suppose we want to substitute **F2** for **F1** in every call of the function **F1**. The sentence like

< Subst21 < F1 e1 > e2 > = < F2 e1 > < Subst21 e2 >

will not work. According to the syntax of Refal, active subexpressions cannot be used in the left side. But even if we extended Refal to allow such a use, the active subexpression **< F2 e1 >** in the right side would be understood as a process, not an object; the evaluation of this call would start before the further evaluation of **Subst21**, even though we did not want it. Likewise, we cannot use free variables as objects, because they are understood by the Refal machine as a signal to make substitution.

The mapping of all Refal object on a subset of object expressions will be referred to as a *metacode*. This mapping must, of course, be *injective*, i.e. the images ('the metacodes') of distinct Refal objects must be distinct, so that there is a unique inverse transformation. We regard the metacode transfor-

mation as the lowering of the metasystem level of the object: indeed, from a controlling device it becomes an object of work. Therefore, when we apply the metacode transformation to an expression E, we shall say that we *downgrade* it to the metacode; applying the inverse transformation we shall say that we *upgrade* E from the metacode. When we actually write Refal programs dealing with Refal programs, we must choose a certain concrete metacode transformation. But when we speak about downgrading and upgrading expressions, we often want a notation allowing us to leave these transformations unspecified. Thus downgrading to the metacode will be represented by a downward arrow, or the underlined combination dn (for 'down'); for upgrading we reserve the upward arrow, or up. When the range of the operation extends to the end of the current subexpression, we simply put dn or up in front of it. If it is necessary to delimit the range, we shall use braces. For example, the concatenation of the downgraded E_1 with the unchanged E_2 is $\{\underline{\text{dn}}\, E_1\}E_2$, while $(\underline{\text{dn}}\, E_1)E_2$ is the same as $(\{\underline{\text{dn}}\, E_1\})E_2$. Obviously, $\underline{\text{up}}\, \underline{\text{dn}}\, E = E$; and if $\underline{\text{up}}\, E$ exists, then $\underline{\text{dn}}\, \underline{\text{up}}\, E = E$.

The purpose of metacoding is to map activation brackets and free variables on object expressions. It would be nice to leave object expressions unchanged in the metacode transformation. Unfortunately, this is impossible, because of the requirement of injectivity. Indeed, suppose that we have such a metacode. Then dn dn **e1** must be equal to dn **e1**, because dn **e1** is an object expression. It follows that two different objects, **e1** and dn **e1**, have identical metacodes.

We can, though, minimize the difference between an object expression and its metacode. The metacode we are using in this book singles out one symbol, namely the asterisk '*', which changes in the metacode transformation. All other symbols and object brackets (parentheses) are mapped into themselves. The following table defines our metacode when used on Refal expressions:

Expression E	its metacode $\underline{\text{dn}}\, E$
s*I*	* **S** *I*
t*I*	* **T** *I*
e*I*	* **E** *I*
$<F\,E>$	$*(F\ \underline{\text{dn}}\, E)$
(E)	$(\underline{\text{dn}}\, E)$
E_1E_2	$\{\underline{\text{dn}}\, E_1\}\underline{\text{dn}}\, E_2$
*	* **V**
S (distinct from *)	*S*

As mentioned before, we use Refal programs as objects of work in the form of Refal graphs. The braces we use in graphs will not be considered formally different from the usual parentheses; this distinction serves only the purpose of visual convenience. In programming, when we need to distinguish between these two kinds of brackets for the reasons of syntax, we pu tin front of the left bracket a characteristic symbol; thus $\{E\}$ will be replaced by **Braces**(E), or **Graph-sum**(E), or just G(E), etc. The same technique is used for other varieties of brackets, like **Begin** and **End** in programming languages, etc.

When the metacode transformation is applied to an object expression, its result can be computed by calling the Refal function **Dn**:

```
Dn {* e1  =  * V <Dn e1>;
    s2 e1  =  s2 <Dn e1>;
    (e2)e1  =  (<Dn e2>) <Dn e1>;
     = ; }
```

For any object expression *E*

$$\langle \mathbf{Dn}\ E \rangle \ == \ \underline{\mathbf{dn}}\ E$$

The inverse to **Dn** function is **Up**:

```
Up {* V e1  =  V <Up e1>;
    *(sF e1)e2  =  <Mu sF e1> <Up e2>;
    * e1  =  <Undefined>;
    s2 e1  =  s2 <Up e1>;
    (e2)e1  =  (<Up e2>) <Up e1>
     = ; }
```

This function, unlike **Dn**, is not applicable to any object expression, but only to such that could be obtained by downgrading a ground expression. The asterisk may be followed only by **V** or a left parenthesis. The function **Undefined** can be thought of as just non-existing in the program, so that when it is called the Refal machine comes to an abnormal stop. In programming, the Refal system will not allow us to call an undefined function. Thus **Undefined** must be given any definition with which the empty argument will cause an abnormal stop, e.g.

```
Undefined { X  =  X; }
```

If *E* is in the domain of **Up**, the following relation holds:

$$\langle \mathbf{Up}\ E \rangle \ == \ \underline{\mathbf{up}}\ E$$

Unlike the analogous relation for **Dn**, this is a relation between *processes*, not object expressions. *E* here can be the metacode of any *ground*, not necessarily *object* (i.e. passive ground) expression. Let E^{gr} be an active ground expression. It stands for a process. We downgrade it to an object expression $E = \underline{\mathbf{dn}}\ E^{gr}$. Then we can upgrade *E* back to the process E^{gr}. The evaluation of $\langle \mathbf{Up}\ E \rangle$ will be a simulation of the evaluation of E^{gr}, as one can see from the definition of **Up**. This function converts the 'frozen' function calls in *E* into their active form, and does this in exactly the same order as the Refal machine -- the inside-out order. Thus we have the relation:

$$\langle \mathbf{Up}\ \underline{\mathbf{dn}}\ E^{gr} \rangle \ == \ E^{gr}$$

The difference between dn and up, on one hand, and **Dn** and **Up**, on the other hand, must be kept in mind. The former are just metasymbols used to denote certain Refal objects; they are not Refal objects themselves. The latter are function names of regular Refal functions.

The time required for downgrading or upgrading an object expression is proportional to its length. It is often desirable to delay the actual metacoding of an expression till the moment when its downgraded form is, in fact, used, because it well may happen that the whole expression or its part will be later upgraded back to its origianl self. Then by delaying the metacoding we would avoid the unnecessary two-way transformation. We shall use the expression

$$*!(E^{obj})$$

to represent a delayed metacoding. This does not violate the uniqueness of the inverse transformation **Up**, because a new symbol, '!', follows the asterisk. For **Up** to handle delayed metacoding, we must add to its definition, in an appropriate position, the sentence:

< Up *!(e1) e2 > = e1 < Up e2 >