Metavariables: their Implementation and Use in Program Transformation

Valentin F. Turchin
The City College of New York
Andrei P. Nemytykh
Programming Systems Institute, Pereslavl-Zalesski, Russia

1 Introduction

Since the discovery that self-application of a partial evaluator can lead to an automatic creation of compilers [2], [13], [7], self-applicability became one of the criteria of the usefulness of program transformers. Self-applicability, however, is but a special case of a more general phenomenon of *metasystem transition*, or *MST* [12] when a computing machine (or a cybernetic system of any kind) becomes an object of control and manipulation by another computing machine (cybernetic system). We are mainly interested in the case where this control and manipulation produces as its final result a program which is equivalent to the original machine, but is "better" in some sense – usually, works faster.

Repeated metasystem transitions create a hierarchy of machines where a machine at the level $n+1$ is a metasystem with regard to the machine(s) on the level n. Several papers appeared lately where a metasystem transition which is not self-application was used to improve the techniques of program transformation ([16], [5]) and multiple metasystem transitions were explored ([3], [4], [6], [1]).

In this paper we consider metasystem hierarchies of abstract computing machines and propose a notation for them: *MST schemes*, also referred to as *stairway schemes*. Our main concern is to develop techniques for efficient computation at all levels of hierarchy, jumping from one level to another when this is possible and necessary for efficient computation. The techniques we have developed were implemented in the programming language Refal-5 [15]; a supercompiler [11], [14] was used as the program transformer. We want to stress, however, that our techniques are language independent. We

present them in the context of a certain language BFL (for 'Basic Functional Language') which is not completely specified. What we need of the language is only the notation of data structures and function calls. With the language fixed, the hierarchy of computing machines becomes a hierarchy of programs for identical machines which execute programs in that language. What the program of the level $n+1$ does with the program on the level n may vary. Equivalent transformation is the main case. But this also can be interpretation (we give examples in Section 5), inversion, or some of more special problems, such as trying to determine if a given function can ever take a given value.

Speaking of *program* transformation we also speak of *function* transformation – we shall not make difference between the two. Function transformers and transformers of function transformers are known as higher-order functions, and there is a huge literature on this concept. The reader may be surprized to see in this paper no references to this literature. This is because our goals are quite different. Usually, people want higher-order functions in order to enhance programming, which creates a need for programming languages which include this feature. Our aim in this paper is not to enhance programming, but to explore the operation of a hierarchy of computing machines each of which understands a fixed programming language, BFL, which, on top of it, is not fully specified.

Actually, we completely ignore the usual set-theoretical definition of a function as a subset of a certain Cartesian product. With our computational approach to the problem, a function is a program in some fixed language — a string of characters which triggers a certain process of computation. With the set-theoretical definition, we have a hierarchy of functions where each next function type makes transformation of functions of the preceding type. If, e.g., the first-order functions are from N to N, then we have the hierarchy:

$$\begin{array}{c} \ldots etc \\ ((N \to N) \to (N \to N)) \to ((N \to N) \to (N \to N)) \\ (N \to N) \to (N \to N) \\ N \to N \end{array}$$

We presented this hierarchy as growing upward, in order to conform with our definitions of the MST hierarchy. Nevertheless, it is *not the same* as our MST hierarchy. It is a hierarchy of set-theoretical function types, while our hierarchy is a hierarchy of the actual use of functions. In our approach, a "first-order" function is, strictly speaking, *any* function, but in practice, a

function which is not a function transformer. As for higher-order functions, i.e. function transformers, they all are of the same type. If we have two transformers, say F and G, nothing prevents us from using F to transform G, in which case F will be on top of G in the hierarchy, or using G to transform F, in which case the order will be opposite. No such tricks can be played on higher-order functions defined set-theoretically.

In accordance with this constructivist approach, we consider symbols which are manipulated by the computing machine as just small pieces of matter, some of which are treated as identical. The variables which we see in computer programs are also nothing else but small pieces of matter different from symbols, but also used in a definite way by the computing machine. Then *metavariables* are symbols, or combination of symbols, of one machine which represent the variables of another machine which is found higher or lower in the MST hierarchy. The subject of the present paper is how to treat these representations for efficient computation at all levels.

In many cases a program transformer must transform a function call which, in fact, can be fully or partially evaluated. Computation of a function call from the next metasystem level is program interpretation, and it is much slower than a direct execution of computation by the machine itself. One of our goals is to develop a method which would allow the system to jump from one level to another, in order to make computation direct whenever this is possible.

In Section 2 we define the first-order language BFL, which is our notation of data structures and function calls. In Section 3 the basic concepts of metacode and metavariables are defined. In Section 4 we define the notation of MST schemes and the notion of an elevated variable. We further examine how it is possible to jump automatically from one MST level to another in pursuit of efficient computation. We find that we must introduce such objects as metavriables of negative degrees, even though by the initial definition the degree can be only positive. In Section 5 we give account of an implementation of our system in the computer. We find that allowing metasystem jumps can radically speed up the computation in program transformation. Finally, Section 6 sums up the results of the paper.

2 First-order language

In this section we define the Basic Functional Language (BFL) which is not, in fact a full programming language, because we leave both the syntax and

semantics of programs not fully defined. For our purposes we only need a notation for data structures and function calls; the rest remains unspecified for the sake of generality of our results.

A *symbol* in BFL is a syntactic element of data structures which is always treated as a whole. We leave unspecified the exact types of symbols, because they may be different in different languages. The minimal set of types should include:

- symbolic names;
- characters;
- whole numbers.

A symbolic name, also referred to as an *identifier* is a sequence of characters which starts with a letter and can include letters, digits and hyphens -. A symbolic name used as a symbol of BFL must start with a capital letter.

All printable characters can be used as symbols. To distinguish character-symbols from symbolic names, we put the former in quotes ' and '. Blanks, if not quoted, are used to separate lexical units whenever necessary.

We need not put any restrictions on the textual representation of numbers.

The symbols of BFL correspond to atoms in Lisp. We need objects which would correspond to Lisp's s-expressions (lists, in particular). We shall call these objects just *expressions*. However, we want to preserve more generality in defining the basic data structures than we see in Lisp. We are interested, of course, in symbol manipulation data structures. All programming languages have their equivalents of symbols, but they differ in the ways of combining them into composite structures. In Lisp and most functional languages composite structures (lists) are binary trees. We shall make it possible to construct trees of arbitrary arity at each node, as well as to use another familiar data structure: a string of symbols which can be read in both directions (unlike Lisp's lists). We achieve this by introducing two basic constructions instead of one: concatenation and enclosure in parenthesis. A BFL's expression can be defined as a sequence of symbols and parentheses where parentheses are correctly paired according to the well-known rules. More formally: an expression is a sequence of *terms*, while a term is either one symbol or an expression in parentheses. Now, `'a+b'` is a string (sequence) of three symbols: `'a'`, `'+'` and `'b'`. A tree with the subtrees $E_1, E_2, \ldots E_n$ can be represented in BFL as:

$$((E_1)(E_2) \ldots (E_n))$$

If we drop the outermost parentheses, we have a *forest.*

The number of constituting terms in an expression may be 0; such an expression is an empty string, just nothing. When the use of an empty string may lead to inconvenience or ambiguity, we shall use the notation [] for it, so that `'a'[]` is the same as `'a'`.

An expression, as we have defined it, is a data structure, a passive object on which the basic computing machine will work. We shall call such expressions *object expressions* to distinguish them from more general BFL expressions which may include variables and function calls. Parentheses give structure to the BFL objects, so we shall often call them *structure brackets* as distinguished from *evaluation* brackets which we shall introduce in a moment.

In accordance with the syntax of its data structures, BFL has two kinds of variables: *symbol variables* and *expression variables*; in short, *s-variables* and *e-variables*, respectively. S-variables take single symbols as their values; e-variables may have as its value any expression. S-variables are represented by the prefix `s.` followed by its name which will be referred as the *index.* An index is either a lower-case symbolic name or a number, *e.g.* `s.1`, or `s.end`. An e-variable is denoted by a prefix `e.` followed by an index, *e.g.* `e.1`, or `e.arg-5`. As a kind of syntax sugar, we can drop the prefix `e.` of an e-variable if the index is a symbolic name: `arg-5`. Such a variable is distinguished from a symbolic name representing a BFL *symbol* by starting with a lower case letter.

BFL's variables may be used as terms in an expression; such an expression will be referred to as a *pattern expression*, or just *pattern.* In a pattern expression, variables are understood as *free*, i.e. free for the substitution of any syntactically admissible value. So, the pattern `'a'e.tail` can be seen as representing the set of all object expressions which start with `'a'`, and `e.1(e.2)` as the set of object expressions which end with a right parenthesis. If there are several entries of the same variable in a pattern expression, they all must take the same value in substitution.

Evaluation brackets < ... > are used to form function calls. <*F E*> is a call of function F with the argument E, where F is a symbolic name or an s-variable, and E a general BFL expression which can include both variables and function calls.

Formally, all functions defined in BFL are functions of one argument, an object expression. But this argument may have a definite *format*, common to all calls of the function. For instance, it can be <F (E_1) E_2>, where E_1 and E_2 are some expressions. Then <F (E_1) E_2> can be seen as a function F of two variables, with the values E_1 and E_2. Or we could use the formats <F $E_1(E_2)$>, or <F $(E_1)(E_2)$>, with the same effect: a matter of taste.

As a syntax sugar, we use commas to separate subexpressions in the argument. To unfold this notation, replace each comma with the inverted pair)(, then add a left parenthesis at the beginning, and a right parenthesis at the end of the argument. Thus, `<F x,y,z>` stands for `<F (x)(y)(z)>` (which, in its turn, stands for `<F (e.x)(e.y)(e.z)>`).

Summary of the syntax of BFL expressions

- An *expression* is a sequence of *terms*. A *term* is one of the following: a symbol, a variable, (E), $\langle E \rangle$, where E is an expression.
- An *object expression* may include only symbols and parentheses (structure brackets). The domain and range of BFL functions are sets of object expressions.
- A *pattern expression* may include symbols, parentheses and free variables, but no evaluation brackets.
- A *general expression* may include all elements listed above.
- An expression is *active*, if it includes at least one pair of evaluation brackets; otherwise it is *passive*.

The BFL notation is, in essence, the one used in Refal. We abstract ourselves from the syntax of BFL programs and the way the BFL machine operates. But to read the BFL notation is, in fact, sufficient for being able to read Refal programs in the simplest version of this language. Here is an example. Let function `Pal` be a recursive predicate which tells if its argument is a palindrom. The following is the definition of `Pal` in Refal (`/*` and `*/` enclose comments):

```
<Pal []> = True                    /* empty string */
<Pal s.1> = True                   /* one symbol string */
<Pal s.1 x s.1> = <Pal x>          /* the same symbol at the
                                      beginning and end */
<Pal x> = False                    /* none of the above */
```

3 Program transformation

When we write a program P' which transforms other programs, P_1, P_2 ... *etc*, we make a *metasystem transition*. The BFL machine M' which runs P' is a metasystem with respect to the machine M which runs P_1, P_2 ... *etc*. Program transformation, with the exception of trivial syntactic changes, is supposed to be based on the *meaning* of programs, and their meaning is defined by the work of the machine which executes them. To make meaningful program transformations, the machine M' must, in one way or another, observe, analyze, control and modify the operation of the machine M which runs the programs being transformed. That kind of a system is referred to as a metasystem.

The metasystem transition from a BFL machine M that executes some programs to a BFL machine M' that transforms those program can be repeated any number of times, producing a *metasystem stairway*:

$$M,\ M',\ M'',\ M'''\ldots$$

We shall assign whole numbers to the *levels* of this stairway; if the level of M is n, then the level of M' is $n+1$.

The domains of functions defined in BFL are sets of *object expressions*. BFL programs, however, may use most general BFL expressions, which include evaluation brackets and free variables. Hence we cannot directly write BFL programs which manipulate BFL programs. To do this we must map the set of general BFL expressions on the set of object expressions and use the images of "hot" objects, i.e. free variables and evaluation brackets, instead of the objects themselves.

We shall call this mapping a *metacode*, and denote the metacode transformation of E as $\mu\{E\}$. Obviously, metacoding must have a unique inverse transformation, *demetacoding*, so it must be injective: it is required that:

$$\forall(E_1, E_2)\ E_1 \neq E_2 \rightarrow \mu\{E_1\} \neq \mu\{E_2\}$$

For convenience of reading metacoded expressions we require that $\mu\{E_1E_2\} = \mu\{E_1\}\mu\{E_2\}$.

Using metacode transformation we can model the behavior of one BFL machine on another BFL machine. We can write a BFL interpreter `Int` (often referred to as a *meta-interpreter*) such that if P is a BFL program defining function F and

$$\text{<F } E_{arg}\text{>} = E_{value}$$

then

$$\texttt{<Int } \mu\{\texttt{<F } E_{arg}\texttt{>}\}, \mu\{P\}\texttt{>} = \mu\{E_{value}\}$$

One can invent any number of metacodes, but for working with metacoded expressions conveniently, it is highly desirable that the image of an object expression be as close to the expression itself as possible. It would be nice, of course, to leave all object expressions unaltered under the metacode, but this is, unfortunately, impossible, because it contradicts to the requirement of injectivity. Indeed, suppose that $\mu\{E_o\} = E_o$ for any object expression E_o. Take a general expression E_g which is not an object expression. Its metacode $E_g{}' = \mu\{E_g\}$ is an object expression. Therefore, $\mu\{E_g{}'\} = E_g{}'$, and $E_g{}'$ is the image of both E_g and $E_g{}'$, which are distinct. This violates injectivity.

We distinguish two types of metacoding: *internal* and *external.*

3.1 Internal metacode

Let S_{ob} be the set of all object expressions, and S_{va} the set of BFL expressions which are not object expression, i.e. include at least one variable or active subexpression (a function call). So, the total set of expressions is $S_{ob} \cup S_{va}$. An internal metacode is defined as

$$\mu : S_{ob} \cup S_{va} \rightarrow S_{ob}$$

in accordance with the above definition of metacode tranformation. An internal metacode requires no extension of the set of object expressions we find in the basic first-order language; the images of expressions from S_{va} are not different from expressions used for any other purpose. The advantage of this method of metacoding is that it requires no alteration of the first-order language. One may be using a familiar language with a well established compiler, and be in no position to alter it as required by the *external* metacoding described in the next section.

One convenient metacode, which is used in the latest implementation of Refal, is defined by Table 1, where S is any symbol, and i the index of a variable.

This metacode leaves strings of symbols unaltered, but adds an asterisk to every pair of parentheses. The internal metacode used in [15] takes the opposite approach: parentheses are not changed, but one character, namely `'*'` becomes `'*V'` when metacoded.

E	$\mu\{E\}$
S	S
s.i	('s'i)
e.i	('e'i)
(E)	('*'$\mu\{E\}$)
<E>	('!'$\mu\{E\}$)

Table 1: An internal metacode.

A few examples of metacoding:

$$
\begin{aligned}
\mu\{\texttt{e.x}\} &= \texttt{('e'X)} \\
\mu\{\texttt{Begin(s.1)}\} &= \texttt{Begin('*'('s'1))} \\
\mu\{\texttt{<Fun (<Fun (25)>)>}\} &= \texttt{('!'Fun('*'('!'Fun ('*'25))))} \\
\mu^2\{\texttt{e.3 <F (A) B>}\} &= \mu\{\texttt{('e'3)('!'F ('*'A) B)}\} \\
&= \texttt{('*e'3)('*!'F ('**'A) B)}
\end{aligned}
$$

When the metacode transformation is applied to an object expression, its result can be computed by a function definable in BFL. We shall call this function Dn (read: Down). It simply adds an asterisk after every left parenthesis. In Refal the definition of Dn is:

```
<Dn (x) y> = ('*'<Dn x>) <Dn y>
<Dn s.1 x> = s.1 <Dn x>
<Dn [] > = []
```

The inverse function Up demetacodes an object expression (when this is possible) by taking away one asterisk following after each parenthesis:

```
<Up ('*'x) y> = (<Dn x>) <Dn y>
<Up s.1 x> = s.1 <Up x>
<Up [] > = []
```

If Up meets an expression which could not have been formed by metacoding an object expression (*e.g.* if a left parenthesis is followed by 's' or 'e'), then an abnormal stop takes place: the function is undefined.

Note the difference between <Dn E> and $\mu\{E\}$. The former is an active Refal expression, the latter is not part of Refal but part of a metalanguage: a notation we use to speak about Refal expressions.

Although the result of metacoding is always an object expression and thus includes no free variables, the programmer can create patterns which express abstraction from some features of the expressions subject to metacoding. Thus, `('e's.i)` is the metacode of an e-variable with *some* index `s.i`. This idea could not be expressed in BFL without metacoding.

3.2 External metacode

We call a metacode *external* if it requires an extension of the set S_{ob} of object expressions of the basic first-order language. Let the extended set of object expressions, which includes all results of metacoding, be S_{mob}. The following properties are required:

$$\forall exp(exp \in S_{ob} \rightarrow \mu\{exp\} = exp)$$
$$\mu\{S_{va}\} \subseteq S_{mob}$$
$$\mu\{S_{mob}\} \subseteq S_{mob}$$

Such a metacode does not change those object expressions which belong to the original first-order language, but the price paid for that is an extension of this set by including images of metacoding, which now must be declared object expressions (otherwise we would not be able to deal with them).

Below we describe the external metacode implemented in the language Flac, which is, basically, a version of Refal specialized for algebraic computation [9]. The reason for using an external metacode is that it makes it possible to jump between the levels of a metasystem stairway more efficiently, as discussed in Sec. 5. The Flac metacode is summarized in Table 2.

The set of Refal expressions is extended by including a potentially infinite set of special signs called *metacoders*. They differ from each other by their *degree*, which is a whole number – positive, zero, or negative. In this paper a metacoder of the degree d will be denoted as $\#^d$. A metacoder, on its own, is *not* a legitimate Refal symbol. It makes an object expression only if immediately followed by a left parenthesis. The syntax of extended object expressions can be described by adding one more type of a term:

$$term ::= \#^d(expression)$$

As one can see from Table 2, metacoding raises the degree of each metacoder by 1. The expression in the parentheses that follow a metacoder may, in its turn, include metacoders, in which case these metacoders are treated according to the same rule.

E	$\mu\{E\}$
`s.`i	`#`^{1}`('s'`i`)`
`e.`i	`#`^{1}`('e'`i`)`
`<`E`>`	`#`^{1}`('!'`$\mu\{E\}$`)`
`#`^{d}`(`E`)`	`#`$^{d+1}$`(`$\mu\{E\}$`)`
S	S
`(`E`)`	`(`$\mu\{E\}$`)`

Table 2: An external metacode. S is any symbol.

One consequence of the external metacode is that the functions `Dn` and `Up`, which with the internal metacode can be defined in BFL, now must be built into the system. `Dn` adds 1 to the degree of every metacoder, `Up` subtracts 1. Otherwise, both functions leave the argument unchanged.

One can see from Table 2 that only positive values of the degree d can appear from the metacoding of first-order expressions. But we extend the set of metacoders by allowing non-positive degrees d. The meaning and the use of non-positive values of d will be discussed later. Note that in contrast to the internal metacode, the demetacoding function `Up` is always defined. When a free variable, say `e.1`, is metacoded and then demetacoded, the result is not a free variable, but an object expression, namely, `#`^{0}`('e'1)`. Thus we have two representations of the same concept. The form `e.1` is used in programs; the form `#`^{0}`('e'1)` appears in metasystem jumps (see Sec. 4.3).

3.3 Metavariables

A *metavariable* is an object obtained as $\mu^d\{v\}$, where v is a free variable and d a whole number, called its *degree*. Free variables, like `s.5` and `e.x` are metavariables of degree 0. With the external metacode of Table 2 the degree of a metavariable is the degree of its metacoder. With the metacode of Table 1, it is $n+1$, where n is the number of asterisks between the opening parenthesis and the type symbol `'s'` or `'e'`. Thus, `('s'5)` is a symbol metavariable of degree 1; `('**e's.y)` is an expression metavariable of degree 3.

We did not include metavariables of negative degrees in either table defining a metacode, because they are not used in programs, but may appear automatically during the computation, as we discuss in Sec 4.3. Actually,

it is possible to extend the language by allowing metavariables of negative degrees in programs, but we did not see much sense in it at the present stage of research. It is quite possible, though, that such use of metavariables will be found in time.

4 Metasystem stairway

4.1 MST schemes

We start with an example. Consider the well-known procedure of converting an interpreter for some language into a compiler by partial evaluation (see [2], [11], [7]). Let L be an interpreter for some language L written in BFL with the format <L program,data>, where program and data are free variables. In order to use the interpreter we substitute some specific program P and data D for the corresponding variables, then we run the call in the BFL machine. Its result is the result of applying P to D.

Let PE be a partial evaluator for BFL written in BFL and having BFL as the target language, i.e. producing a BFL program at the output. At this time we could use a partial evaluator with any target language L', but as we shall see below, for further construction of metasystem staircases we need that the target language be the same as the input language.

PE requires two arguments: the metacoded function call to evaluate, and the metacoded program defining the function(s) in the call. We apply partial evaluation PE to the call of L where some program P is substituted for program, while data remains free. This call is <L P,data>. Let us use the internal metacode of Table 1. Then the metacode of this call is

$$(\text{`!'L } \mu\{P\},(\text{`e'data}))$$

The metacode of the program P can be obtained by using Dn on P. Thus, the call of PE we want is:

<PE (`!'L <Dn $P$>, (`e'data)), def(L)>

where **def**(L) is the metacoded BFL program which defines L. (A remark on notation: L is an abstract reference to a language; L is an identifier, an arbitrary name for the function which interprets programs like P written in L; **def**(L) is the program in BFL which defines L).

For a clearer view of this expression, we present it as an *MST scheme*[1]:

[1]This notation was first used by one of the present authors (VT) in lectures at the

```
<PE .............,def(L)>
    <L P ,data>
```

Such a scheme is a representation of BFL expressions according to the rule: whenever a subexpression has the form $E_1\mu\{E_2\}E_3$, the metacoded part is moved one level down and replaced by dots on the main level:

$$E_1\mu\{E_2\}E_3 \quad \Longleftrightarrow \quad \begin{matrix} E_1 & \ldots & E_3 \\ & E_2 & \end{matrix}$$

The parts of the overall BFL expression which belong to different meta-system levels are put on different lines. BFL expressions on the bottom level are written the same way as if they were on the top level; metacoding is implicit and is indicated by putting them one line down. To convert a stairway scheme into an executable BFL expression, we must metacode each level as many times as long is its distance from the top. We shall refer to this conversion as the *folding* of an MST scheme.

In our two-level scheme the call of L, which is submitted for partial evaluation by PE, is a function of data only, since the value of `program` is fixed at a specific program P. After PE performs all operations which can be performed because the program P is known, it outputs the residual program which is nothing but the translation of the program P into BFL. Function PE works as a compiler.

4.2 Multilevel MST schemes. Elevated variables

Even though the result of the partial evaluation according to the two-level stairway scheme may be an efficient compiled program, the process of compilation itself is still less efficient than it could be. This process depends on two arguments: the program P and the definition of the language **def**(L). We can make the program variable, while keeping the definition of the language fixed; this will provide a job for the partial evaluator, which in this way will be applied to itself.

How should we modify our scheme? If we simply replace P by the variable `program`:

```
<PE .................,def(L)>
    <L program,data>
```

University of Copenhagen in 1985. Ever since, its various versions were used in seminars on Refal and metacomputation in Moscow and New York. In a published form it first appeared in [3].

we do not get what we want. Here program is treated in the same way as data. Partial evaluation is applied to the function L of two arguments. Therefore, the function which results from PE is, again, a function of two variables, an interpreter of L.

The value of program, even though it is an argument of L, must be provided on the level of PE, so that when L is running (being driven by PE), the program is fixed. We represent this situation by raising program to the top level, and leaving the bullet • in the place where this variable originated on the bottom level:

```
<PE .. program .....,def(L)>
    <L    •    ,data>
```

We shall call such variables as program *elevated.* Even though program is used by L, it is not free for it; it is free on the level of the partial evaluation function PE; to run PE we must first substitute some specific program for program. Hence L always receives a fixed program. The result of PE will be a transformed (partially evaluated) function L, which now depends only on the variable data.

The formal meaning of an MST scheme is defined, of course, by its conversion into a normal, executable BFL expression. The way elevated variables must be converted can be seen from the following reasoning. Let program take some specific value P. It is positioned as indicated by the sign •, i.e. on the bottom level. When we fold the MST scheme into a BFL expression, P will become $\mu\{P\}$, which may be computed by <Dn P>. Thus, program must be finally replaced by <Dn program>. The stairway scheme converts to the expression:

```
<PE ('!'L <Dn program>, ('e'data)), def(L)>
```

This function call includes as one of its arguments an object expression that is fixed and may be quite bulky, namely, def(L). Hence there is a good chance that partial evaluation of PE will produce a more efficient program than when this function is run directly in the BFL machine. So we submit this function, without any alterations, to PE:

```
<PE ..............................,def(PE)>
    <PE .. program ......,def(L)>
        <L    •    ,data>
```

The folding of this stairway scheme into an expression is:

```
<PE ('!'PE ('*!'L ('!'Dn ('e'program)),
                  ('*e'data)), <Dn def(L)>), def(PE)>
```

We see here that those parts of the expression which come from the bottom level, such as the variable `data`, are metacoded twice. This call of PE, when executed, produces a program which is a (compiled) compiler for the language L.

Constants (object expressions) also can be raised or lowered in MST schemes. When we raise a constant in the same manner as we raise variables, i.e. leaving a bullet • in the place of origination, then we simply copy it and put at the desired level. These two fragments of MST schemes are equivalent:

```
.....
....const ..
```

and

```
....const ...
..... • ....
```

Indeed, how do we execute the second scheme? We take *const*, metacode it, and put in the place of origination marked by the bullet. In the first scheme *const* is already in the same place, and and its degree in metacode is one unit greater than if it were on the upper level. Thus the schemes are equivalent.

We also can move constants without leaving the bullet, but then we must not forget to adjust the metacode degree. When raised to the next upper level, the constant must be metacoded, because it loses one degree of implied metacoding. The fragment

```
.....
....  const ....
```

is equivalent to

```
..  μ{const}...
.............
```

A shorter version of the MST notation is often preferable. Instead of putting explicitly the definition of the function being transformed as one of the arguments of the transformer, we can understand the name of the function as the program which defines it. This is how the three-level scheme above looks under this agreement:

```
<PE .....................>
    <PE .. program .......>
        <L    •    , data>
```

Whenever the definition of a function being transformed is absent from the MST scheme, it should be understood that the function name is standing for it.

4.3 Jumping between metasystem levels

It often happens that a program transformer must transform a function call which, in fact, can be simply evaluated. The argument may include no free variables or yet uncomputed function calls or, if there are some, they may not be consulted at any stage of evaluation. Even more frequent is a situation where such independence of unknown data holds for a part of the evaluation process, even though not for the whole length of it.

Consider our two-level scheme of compilation:

```
<PE .. program ...,def(L)>
    <L    •   ,data>
```

The interpreter L operates on a known program and unknown data. On some stretches of computation L will work on the program, but without consulting the data. An obvious example is syntax parsing of the program. Further, if the language L includes go-to statements with jumps to a label, then it may be necessary to examine a big piece of program in search of the needed label. The work of the function PE in this part of computation will be nothing else but simulation of the work of L, which, of course, will take much more time (sometimes, by two orders of magnitude), than a direct run of the function L.

Consider the evaluation of PE when a certain program P is given to it as the value of program:

```
<PE ... P.....,def(L)>
    <L  •  , data>
```

The implementation of BFL must be able to demetacode the bottom level of the MST scheme converting an object expression to a function call, and pass control to this function call. The bottom level of our scheme is (we use Table 1):

$$(\text{`!'L } \mu\{P\} \text{ , (`e'data)})$$

After demetacoding:

$$\text{<L } P \text{ , } \mu^{-1}\{(\text{`e'data})\}\text{>}$$

How should the BFL machine handle $\mu^{-1}\{(\text{`e'data})\}$? Formally it is `e.data`, a free variable. It is not an object expression, so it cannot enter the domain of function L. It is an unknown entity, a "hole" in the data. The BFL machine must carry on computation of L as long as its course does

not depend on the unknown data represented by $\mu^{-1}\{(\text{'e'data})\}$. When such a dependency is discovered, the current configuration on the bottom level must be metacoded and turned back to PE with an indication that computation has been suspended. Then the computation on the top level must be resumed. If the computation was successfully completed without noticing the whole e.data, the value of the call of L must be passed back to the top level with an indication that the computation has been completed. This is what we refer to as *metasystem jumps.*

We observe here two different uses of free variables, which warrants treating them as two distinct kinds of entities: free variables proper, and metavariables of the degree 0. The former are used in programs but cannot be used as data. The latter cannot be used in programs but emerge in demetacoding and are used behind the scene as unknown data, if the implementation of BFL allows for metasystem jumps.

It is easy to see that if we allow for MST schemes of arbitrary height, we need the whole spectrum of metavariables of non-positive degrees, not just of degree 0, because the function which is demetacoded and activated, may, in its turn, cause a jump down and activate a function call lower in the hierarchy.

To be more specific, consider this scheme, where T is some function transformer, and G a function:

```
<PE ............ >
  <T .......e.1 >
    <G 'abc' • >
```

In the strict BFL it is:

```
<PE ('!'T ('*!'G 'abc'('e'1)))>
```

After PE passes control to T the activated call is:

```
<T ('!'G 'abc'e.1) >
```

Since T is also a function transformer, it may pass control even down to G:

$$\texttt{<G 'abc'}\ \mu^{-1}\{\texttt{e.1}\}\texttt{>}$$

Suppose we use only one type of "holes" in data, i.e. identify $\mu^{-1}\{\texttt{e.1}\}$ with e.1. Suppose function G finds that it can do no single step of computation and returns control, with metacoding, to T, and then, with metacoding

again, to PE. Then `e.1` will return as `('*e'1)`, instead of `('e'1)`: an error. If we keep our metavariable as having the degree -1, the result will be correct. Suppose `G` is defined so that

```
<G 'ab' e.x> = e.x
```

Then the call of `G` will be evaluated with the result `'c'`μ^{-1}{`e.1`}. After two metasystem jumps up, the call of `PE` will be evaluated as `'c'('e'1)`. We must not forget that program transformers return *metacoded* programs (and could not do otherwise, because programs in BFL are not object expressions). Therefore, this output must be understood as a program which, given `e.1`, returns `'c'e.1`.

4.4 Relativization of levels

By our initial definition of an MST scheme, its top level is the level on which we have evaluation brackets and free variables; the expressions on the lower levels are all metacoded at least once, so they are object expressions. But with jumps from one level to another a generalization is needed. When the metamachine M' passes control down to M, it is on the level of M that the evaluation takes place, while the upper part of the scheme is temporary ignored. We shall call the *reference level* the level on which the BFL machine is working. Thus, the reference level is the level at which the variables are free and evaluation brackets active. By the *absolute level* of an expression in a scheme we shall understand its distance from the bottom.

If the reference level on the absolute scale is r, and the absolute level of an expression is m_{abs} then $m_{rel} = m_{abs} - r$ will be referred to as the expression's *relative level*. Relative levels below the reference level are negative, above it positive.

Jumps from one level to another are always accompanied by the corresponding metacode transformations. Therefore, the number of times an expression is metacoded in the executable BFL expression represented by an MST scheme, is equal to its distance down from the reference level, i.e. its relative level with the minus sign (this is because we count levels from the bottom up, and degrees of metacoding from the top down).

With a purely functional BFL, it is only the relative level of an expression that matters. We can choose and change the absolute scale at our convenience. This is the principle of invariance with regard to metasystem level: all operations and relations remain the same from one level to another.

When we construct a metasystem staircase, we start with a function call on the ground level which usually is "first-order" function, not a function

transformer. We do not expect to meet in the arguments of such a function any expressions obtained by metacoding variables or function calls.

The function call of such a function as PE which transforms the first-order function may include expressions metacoded once, but no more than once. When we go on constructing the metasystem staircase, each new level increases by one the maximal degree of metacoding. Therefore, the maximal degree of metacoding to be met in the function call represented by the scheme will always be equal to the absolute index m_{abs} of the top level in a staircase scheme. This helps read expressions obtained by folding MST schemes, but does not exclude, with an internal metacode, the possibility that the first-order function can deceive us into thinking that we see a metacode, while in fact we do not. Thus, with the metacode in Table 1, the first-order function may use the combination `('s'1)` for some purpose, not meaning $\mu\{s.1\}$. With an external metacode, though, such deception will not be possible.

4.5 Bound variables and parameters

We refer to metavariables of the relative level -1 as *bound* variables. The meaning of this notion is exactly the same as in mathematics and formal logic: a variable is bound if it runs over its domain of values and thus is not free for substitution; such are the variables of integration in calculus and quantified variables in logic. Indeed, look at the way the variables `program` and `data` are used in the MST scheme of compilation:

```
<PE .. program .....,def(L)>
   <L    •     ,data>
```

Function PE which transforms function L is a function of the second order. When it is active it does not deal with any specific value of `data`, but transforms the whole function L with `data` running over the whole set of its possible values. It is a free variable for (one of the arguments of) function L, but a *bound variable* for PE.

As discussed in Sec. 4.1, for PE to perform partial evaluation with a given program, the variable `program` must be elevated to the level of PE, where it becomes free. Elevated variables are characterized by two levels. We call the *definition level* of a variable the level at which it is free. The *usage level* of a variable is the level where it is used as a variable of some function, i.e. where its value is, normally, examined. In stairway schemes an elevated variable is placed on the definition level. Under it, at one of the lower levels which we refer to as the usage level, a bullet • is placed. The difference

between the definition level and the usage level is the *elevation* h of the variable. In the scheme above the definition level (absolute) of `program` is 1, the usage level is 0, the elevation 1.

More than one entry of the same variable may be found in a stairway scheme. Their usage levels and, therefore, elevations, may be different. The definition level of a variable is, of course, the same for all entries, because identical identifiers on different levels represent different variables. There can be no confusion: variables on different levels take their values at different times.

An elevated variable may be free or bound. In the scheme above, `program` is free. If we add one more level it will be bound with regard to the new top level, but will remain free for the level at which it is defined. All characterizations of variables as free or bound are relative to some reference level. The elevation index is an absolute characteristic, because it is the difference between two levels. It does not change in metacoding and equals 0 for non-elevated variables.

While metavariables of the relative level -1 have been known in mathematics as bound variables, elevated metavraibles represent another familiar concept: they are *parameters*. Indeed, how do we distinguish between a function of two arguments $f(p, x)$ and a family of functions $f_p(x)$ with the parameter p? Computationally, it is the same function in both cases, because both x and p must be given for making the computation. The difference appears only when a higher metasystem level appears. This level may be a human being who, *e.g.* , draws a series of functions of x with a few selected values of p, instead of drawing a two-dimensional surface. A parameter is a variable which is controlled on the metasystem level, as by the person who picks up a value of p, performs computation, then picks another p, *etc.* The same relation we observe in partial evaluation: the variable `program` is free as long as the reference level r is 1; we say that it is free for PE. However, when the reference level becomes 0, actually or for the purpose of analysis and discussion, `program` becomes a parameter; so we say that it is a parameter for L.

To sum up the hierarchy of metavariables:

- The variables on the reference level (relative level 0) are *free*.
- The variables on the relative level -1 are metacoded once. They are *bound* variables.
- The variables on the relative levels below -1 are metacoded at least

twice. As long as there is no metasystem transition to raise them, they are just pieces of data which are compared as ordinary object symbols and expressions.

- The variables of positive relative levels are metavariables *of negative degrees* which emerge in demetacoding. They are not object expressions and cannot be used in programs.

- Elevated variables are *parameters* for the functions that use them;

4.6 The rule of two levels

When functions at all non-zero levels in a staircase scheme are equivalent function transformers, *the rule of two levels* can be used in order to see clearly what to expect from the computation of the expression represented by the scheme. It goes as follows.

The variables on the top level (free) are the variables of the function to compute; some specific values must be substituted for them. The variables on the next level down (bound) are the variables of the function whose definition (the program for it) results from the computation on the top level.

Since the top function is supposed to do an equivalent transformation of the program, the program resulting from computation is equivalent to the definition by the stairway scheme in which the top level is taken off. Therefore, we can again use the rule of two levels on this decapitated scheme, and go on like this until we arrive at the bottom level.

As an example, consider the four-level scheme of compiler compiler (compiled generator of compilers), which is obtained from the three-level scheme of compiler generation by the following procedure: replace the definition of the language **def**(L) (which has been treated as a constant until now) by a variable, then raise that variable to the next level, and add one more PE as the new top level:

```
<PE³ ..........................................,def(PE²)>
   <PE² ....................language,def(PE¹)>
      <PE¹ .. program ......,  •  >
         <L     •     ,data>
```

At the bottom (level 0) we see the goal of computation; all other levels make program transformation using PE, with a superscript indicating the

function's absolute level. The goal is this: given the interpreter of a language L defined by the program def(L), a program P in that language, and a data D for that program, compute the result $P(D)$ of running P on D. Instead of making the computation directly in the interpretation mode, $L(P, D) = P(D)$ we make the following steps.

Step 1. Evaluate the function represented by the scheme. On the top level (level 3) there are no variables; only the definition of PE^2, which is a constant. Therefore, we do this computation once for all times. The result, according to the rule of two levels, is some function which depends on the variable `language`. We denote it as *Compgen*, since, as we shall see below, it is a compiler generator.

Step 2. Compute *Compgen*(def(L)). The result will be the same as that of running PE^2 with `language`=def(L), because PE^3 is an equivalence transformation. It is a function of the variable `program`. We call it *Comp*; It is a compiler for L.

Step 3. Compute *Comp*(P). The result will be the same as that of running PE^1 with `program` = P, because PE^2 is an equivalence transformation. It is a function of `data`, which we shall call *Prog*.

Step 4. Compute *Prog*(D). The result will be the same as running L, because PE^1 is an equivalence transformation, i.e. $L(P, D) = P(D)$. To sum up:

$PE^3() = Compgen$
$Compgen(\mathbf{def}(\mathrm{L})) = Comp$
$Comp(P) = Prog$
$Prog(D) = L(P, D) = P(D)$

With a shorter version of our notation the MST scheme is:

```
<PE³ ..............................>
    <PE²....language..............>
        <PE¹    |    program .....>
             < •       •    ,data>
```

Note that the bullet for `language` on the bottom level is a BFL program, not a BFL program in metacode, as it was in the original MST scheme where `language` was to be replaced by such expressions as def(L), which is, by our definition, the first metacode of the program for L, and it could not be otherwise because it is an argument of the next-level function, hence an object expression. This explains why the variable `language` is now elevated by two levels, while it was elevated by one level in the original version.

4.7 Two ways to treat elevated variables

There are two ways of treating elevated variables. Consider some elevated variable $t.i$ on the level where it is free. A direct approach is to understand $t.i$ as taking any value it may be assigned on the input. Since each step down in the metasystem stairway indicates metacoding, the value of a free variable with the elevation index h must be metacoded h times before being actually used. A free variable $t.i$ with the elevation h is <Dn^h $t.i$>. A metavariable of degree m (relative level $-m$) with elevation h is obtained from this expression by using the metacode m times:

$$\mu^m\{\texttt{<Dn}^h\ t.i\texttt{>}\}$$

For instance, if we use the internal metacode of Table 1, then the metavariable `e.7` of degree 2 with the elevation 3 is rendered by the expression:

```
('*!'Dn ('*!'Dn ('*!'Dn ('*e'7))))
```

We shall say that a variable used like that is *downed at use.*

We can use another method of treating elevated metavariables. Consider again the scheme of compilation by partial evaluation:

```
<PE¹ .. program .....,def(L)>
    <L     •  ,data>
```

It stands for the expression:

```
<PE¹ ('!'L <Dn program>, ('e'data)), def(L)>
```

If we simply submit this expression for transformation by PE^2, then the call of `Dn` will be frozen in the metacode together with the variable and passed further in this form, waiting for the moment when it will be activated (if at all) and used on the value of `program`. This would be what we called above for a variable to be downed at use. Instead, we can make the metacoding of the value of `program` immediately by introducing a new variable which takes on the metacoded value of `program`:

```
program1 = <Dn program>
```

This is the method we have actually used. We call it *downing at input.* Now the call of PE^2 becomes:

```
<PE² .....,def(PE¹)>
    <PE¹ ('!'L program1, ('e'data)), def(L)>
```

There are no references to Dn now, which makes things much simpler. However, in the process of program transformation by PE^2 we must remember that program1 cannot take as its value any expression, but only such that may result from metacoding an *object expression*; in other words, metavariables of degree 1 cannot enter its value. If the metacode of Table 1 is used, this means that every left parenthesis must be immediately followed by an asterisk.

Computation of PE^2 results in a program which defines a certain function Comp1 which depends on program1. If the variable program takes program P as its value, then the value of program1 must be taken as <Dn P>. Then function Comp1 will produce the translation of P:

$$Comp(P) = Comp1(Dn(P)) = Prog$$

In the four-level scheme of Sec. 4.6 we can treat language also as an elevated variable downed at input. Then the use of this scheme can be summarized as follows:

$PE^3() = Compgen1$
$Compgen(\mathbf{def}(\mathrm{L})) = Compgen1(Dn(\mathbf{def}(\mathrm{L}))) = Comp1$
$Comp(P) = Comp1(Dn(P)) = Prog$
$Prog(D) = L(P, D) = P(D)$

Introduction of elevated variables creates a hierarchy of object expressions. Let S^0 be the set of all object expressions. Let S^{h+1} be the set of all expressions which can be obtained by metacoding some expression from S^h. Then

$$S^0 \supset S^1 \supset S^2 \ldots$$

We shall say that an object expression E has the *degree* h if $E \in S^h$ and $E \notin S^{h+1}$. An object expression of degree h can be demetacoded up to h times, but no more if the result is to remain in the set of object expressions. With the internal metacode of Table 1, the degree of an expression can be determined by examining all left parentheses and counting the number of asterisks following each of them. The minimal of these numbers is the degree h. Functions of elevation h have S^h as their domain.

We used renaming of variables and functions as a way to explain the downing at input. Actually, this renaming is not necessary. We extend the metacode, as in Table 1, by providing representation of metavariables with non-zero elevations. In our implementation (see the next chapter) the

metacode of an e-variable with elevation h is represented by inserting h between the index i and the closing parenthesis: $(\texttt{'e'}\,i\,h)$. All functions dealing with an elevated variable must take into account that its domain is not the total set of all object expressions, but only S^h. At the final stage of work we return to each variable, which are now free for substitution, its initial meaning as having its real, *not yet metacoded*, value. Then we must metacode this value h times before applying the function:

$$\text{Replace:} \qquad Func(\ldots var \ldots) \Longrightarrow Func(\ldots Dn^h(var) \ldots)$$

where $Func$ is the function resulting from transformation.

Ignoring elevation of variables, i.e. treating them all as if they were of elevation 0, is an error, because it is assumed in program transformation that a variable can take values which in reality it cannot take. How this error manifests itself depends on the semantics of BFL and its implementation. In special cases it may remain hidden. In our experience, self-application of a program transformer (supercompiler) is impossible without the correct usage of elevated variables. We have done this simple experiment. In a successful self-application of the supercompiler we changed the elevation of a key variable from 1 to 0. Then instead of giving out the correct result, the program went into an apparently infinite loop.

In the early work on partial evaluation by N.Jones with co-workers [8] it was stated that in order to achieve good results in self-application, a preliminary binding time analysis ("off-line") was necessary. However, R.Glück [3] showed that with a correct use of metacoding — which in terms of the present paper amounts to the correct treatment of elevated variables — partial evaluation is self-applicable without the binding time analysis: "on line".

5 Implementation and testing

5.1 Refal-5 and Freezer

We have tested our concept of metavariables and the effect of jumping between metasystem levels on performance time using a supercompiler for flat Refal-5 as the program transformer (the supercompiler for Refal-Flac is still under development). The implementation of Refal-5 (see [15]) was modified to accomodate for a metacode which handles metavariables of different elevations downed at input. This metacode is as in Table 1, except that μ\{e.i\} now is $(\texttt{'e'}\,i\,h)$, where h is the elevation. The metacode μ\{s.i\}

of a symbol variable remains (`'s'`i), because symbols do not change under transformation by this metacode, so all s-variables are treated the same way, independently of their elevations. (Note that the *degree* of an s-variable still makes a difference, since it defines the position of the variable in the MST scheme.)

We have described the metacode as seen by the user, a part of the language. Now we briefly describe the representation of data in the computer. Refal expressions are repesented as doubly linked lists of certain basic structures referred to as *pieces*. A piece stands for one of: a symbol, a structure bracket (parenthesis), an evaluation bracket. A symbol keeps two references: to the preceding and the following piece; brackets have those too, plus a reference to the matching bracket. Evaluation brackets also have a system of references which make up a stack of function calls. The type of a piece is kept in a special field. This data structure represents the work field of the abstract Refal machine.

As long as we do not allow jumping between levels, the above is all we need for execution of Refal programs. Metavariables of non-positive degrees cannot be found in the view field; metavariables of positive degrees are legitimate Refal expressions and are represented as such (recall that this is an *internal* metacode). For instance, the second metacode of `s.5` will be represented by five pieces which make up the expression (`'*s'5`). To make jumping possible, we introduce one more type of a linked piece called an *unknown*. It represents a metavariable of a degree $d \leq 0$. A piece of this type has two references as a symbol and an information field which includes: (1) the type of the variable (`s` or `e`), (2) its index, (3) its elevation, (4) its degree, thus

$$unknown(t, i, h, d) = \mu^{-d}\{t\,.\,i\,.\,h\}$$

In the Refal-5 system the program is first translated into what is referred to as the Refal assembly language, which is then interpreted. The assembly language consists of about twenty basic operations on doubly linked pieces, such as chop off the first or the last symbol of an expression or, conversely, link up a symbol to an expression at its beginning or end. The type of each link is checked in operations, of course. If an unknown piece prevents further execution of the current step of the Refal machine, this step is called off, and a metasystem jump up is performed, as required by our definition of jumps in Section 4.3.

The interaction between metasystem levels takes place through the concept of a *freezer*.

Let a function transformer T, such as PE or the supercompiler, face the task of transforming a call of function F. In the metacode of Table 1 it is (`'!'F` E_g), where E_g is the metacode of the argument of F, which may include free variables and calls of other functions; so it is a general Refal expression. Before doing anything else, T demetacodes the call of F and passes it as the argument to the special function Freezer:

```
<Freezer <Up ('!'F Eg)>>
```

The job of the freezer is to catch the moment when the computation requires the value of an *unknown* and freeze the process, i.e. convert the contents of Freezer into metacode, and pass control back to T. Then T will continue with its method of transformation, knowing that whatever could be done by direct evaluation, has been done.

Freezer is a service function which can be used only in combination with the function Up, as above. Actually, in the current implementation of Refal-5 the user cannot call Freezer, but only the built-in function Ev-met ('evaluate in metacode') which works as if it were defined by the sentence:

```
<Ev-met x> = <Freezer <Up x>>
```

When computation takes place inside of the freezer, its last stage is modified. Instead of just returning the value of the function call as its result, the process ends with one of the following three outputs.

1. The execution reaches a successful end with some result E which is a passive expression (but still may include *unknowns*). Then the call of the freezer is replaced by `0` $\mu\{E\}$.

2. At a certain stage of the execution, when the expression in the freezer is E, the next step becomes impossible either because of an interfering *unknown*, or because E is a call of the special function Residue, which must not be computed, but translated into a call of some residual function in the final program. Then a *freeze* takes place. First the enclosing call of Freezer is found. If there are more than one, the innermost call is taken; if there are none, an error condition occurs. Then the call of the freezer is replaced by: `1` $\mu\{E\}$.

3. At a certain stage of the execution, when the expression in the freezer is E, an abnormal stop (recognition impossible) occurs. Then the call of the freezer is replaced by: `2` $\mu\{E\}$.

problem	no jumps	jumps	speed-up
1. Pat	7	5	1.4
2. Spas	17	2.5	6.8
3. Metfab	13	3	4.3
4. Metpat	112	14	8
5. Metpat1	747	33	23

Table 3: The effect of metasystem jumping.

The described method of doing partial evaluation at every metasystem level has one drawback: before treating each of the function calls, demetacoding is necessary, and it is followed by metacoding after the end of evaluation. Both procedures are linear with the size of the processed expressions, and this work is done even in those cases when it turns out that not a single step of computation can be made.

In Section 5.3 we discuss how this drawback can be eliminated using an external metacode.

5.2 Effect of jumping

The use of jumping may result in a very significant gain in efficiency when partial evaluation is a substantial part of the job assigned to the program transformer, as we illustrate by the following examples.

Table 3 presents the computation times (in seconds) of running the supercompiler on PC/486 in two versions: without (col.2) and with (col.3) metasystem jumps; col.4 records the speed-up factor.

In line 1 the problem 'Pat' was to make the transformation defined by the following MST scheme:

```
<Scp ..................  >
    <Patrec 'aabaac', e.s>
```

where `<Patrec p,s>` is a straightforward pattern recognition function; it finds a pattern p in a string s by comparing p with the prefix of s and shifting the pattern by one symbol when a disagreement is found. Scp is the supercompiler. The result of transformation is a pattern recognizer specialized to look for the fixed pattern `'aabaac'` in an arbitrary string s. In this way we obtain automatically the algorithm of string pattern matching described in [10]. Here we see only a small speed-up.

Problem 2 is a compilation by partial evaluation which we discussed above. In the role of PE we have the supercompiler Scp, and the language L is a language we have called Spas (Small PAScal). It includes reals and integers, arithmetic and logical expressions, assignments, if-then conditionals and while loops. The program compiled was iterative factorial. Here we have a significant speed-up, it runs almost 7 times faster.

Problem 3 ('Metfab': computation of Fab in the metasystem) is a three-level scheme:

```
<Scp ......... >
   <Int .. x  >
      <Fab • >
```

Here Fab is a simple function which changes, in its argument, every 'a' into 'b'; Function Int is an interpreter of the language in which Fab is written. The variable x is *elevated* by one level. If we take off the top level in the scheme, what remains would describe the computation of <Fab x> with an arbitrary x by interpretation of the program for Fab. This function is equivalent to Fab but works much slower. Transforming it by Scp returns an equivalent efficient program, which is, as it should be expected, an exact copy of the original program for Fab.

Problem 4 is similar to 'Metfab' but here the bottom-level program is pattern recognizer and has two variables:

```
<Scp .............. >
   <Int ..... p  s >
      <Patrec •, •>
```

The result, again, reproduces the initial definition of Patrec

The greatest gain in speed was achieved when interpretation and partial evaluation were combined in Problem 5:

```
<Scp ...................... >
   <Int ..... 'aabaac'  s  >
      <Patrec    •    , • >
```

(we dropped here the metacode transformation of the constant 'aabaac' because symbols do not change in the metacode used). The result here is the efficient pattern recognizer specialized for 'aabaac'.

We have also tested our system in a successful self-application of the supercompiler, but this will be described elsewhere.

5.3 Making degrees relative

Doing program transformation it often makes sense to use for object programs a maximally simplified programming language, into which programs written in a more convenient language can be automatically translated. Even though Refal is a pretty simple language, our latest version of the supercompiler works with programs written in a subset of Refal we refer to as *flat* Refal. Generally, we call a language flat if it does not include syntactic means to directly represent the composition of functions. Flat Refal allows only such sentences the right side of which is either passive (no function calls) or consists of a single function call with a passive argument (no nested calls). Still flat Refal is universal, which easy to understand recalling that the language of Turing machines is also flat. Moreover there is a simple algorithm which translates a program from the basic version of Refal into a flat form, and there is only a limited growth of volume in this translation: the flat language remains practicable, if not very practical.

There is something else, though, that we allow in flat Refal: calls of *external* functions, i.e. functions which are not subject to transformation. They break down in two kinds: *computable* and *residual*. Calls of computable functions are evaluated in the course of program transformation; they may be nested inside transformable functions or, at any depth, inside their own kind. Residual function calls move into the output program. We do not go here into technical details concerning external functions.

In this section we present an idea of how to further improve the technique of metasystem jumping, limiting ourselves to such languages as flat Refal.

As mentioned in Sec.5.1, the method of jumping between metasystem levels described there requires a pass through the whole function call during demetacoding, and then a pass through the result to convert it back to metacode. Apparently, this is inevitable if an internal metacode is used. Using an external metacode and restricting the language to flat Refal way, we can avoid these passes, which makes jumping between levels more efficient.

We use the metacode of Table 2 extended by addition of metacoders of negative degreees. At any moment in time the *reference degree* d_{ref} for metacoders is being kept. It is a whole number which is equal to the distance from the top of the stairway down to the current reference level. We also can put it in this way: since the degrees of metacoders are also counted down from the top level, d_{ref} is the degree of the metacoders on the reference level. Now, the treatment of a metacoder is based not on its degree d, but on its relative degree with regard to the reference level:

$d_{rel} = d - d_{ref}$. Instead of increasing the absolute degrees of all metacoders in the metacode transformation, we simply decrease the reference degree d_{ref}, which raises the reference level. To demetacode an expression we have, according to our original definition, to decrease all degrees by one. Instead of that, we increase d_{ref}, i.e. lower the reference level.

When a program in Refal is written, the reference degree is, by definition, 0. Therefore, the absolute degrees of metacoders are at the same time their relative degrees. Accordingly, the initial value of the reference degree d_{ref} is always 0.

Except for making metacoder degrees relative, the jumps up and down the metasystem stairway remain the same as described in Sec. 4.3; control can freely travel throughout all levels. The cost of a metasystem transition is constant and negligeable.

It should be noted that without the restriction to flat BFL, it is, apparently, impossible to completely avoid the examination of the expression in the freezer. This is due to the fact that when a function call <F E> is demetacoded, E may include nested function calls. In the implementation of Refal-5, control over evaluation of functions is carried out, as usual, by maintaining a stack. When a function call with nestings given in the metacoded form is demetacoded, it is converted into a piece of stack which is added to the main stack of the Refal system. This allows to start evaluation in the applicative order, as required by the semantics of Refal-5. Conversely, when the result is metacoded, and is still active, the function calls which were put in stack after `Freezer` are taken from the stack and converted into an expression. Relative degrees of metacoders do nothing to avoid this expense.

6 Conclusion

1. We have considered hierarchies of abstract computing devices, *BFL machines*, where a machine on the level n is examined, run, modified, or otherwise controlled by a machine on the level $n+1$. We refer to the construction of each next level in such a hierarchy as a metasystem transition. In this view of computation, variables are but certain elements, details of the machinery which take part in computation playing their specific roles.

2. We have introduced the concept of a *metavariable* as a constant representative of a variable of one machine in another machine on a different metasystem level. A metavariable is characterized by a whole number called

its degree d. A metavariable with $d = 0$ is the usual free variable. A metavariable with a positive degree represents a variable of a machine on a lower level; if a variable's degeree is negative it represents a variable of a machine which is higher in the hierarchy.

3. A *metacode* is some mapping of a machine at a certain level n onto the data structures of the machine at the level $n+1$. We have distinguished two kinds of metacode: internal, which does not require any changes in the language as seen by the user (implementation needs adjustment, though, if we want to use metasystem jumps); and external, which introduces new features to the language. The programs in which we implemented out techniques were using an internal metacode.

4. We have introduced *MST schemes* which use the vertical dimension in order to provide a convenient visual representation of the hierarchy of computing machines.

5. We introduced a hierarchy of *elevated* (meta)variables which have domains restricted by repeated metacoding. We have shown that for a successful self-application of a supercompiler it is necessary to take into account that variables may have different elevations.

6. It often happens that a program transformer must transform a function call which, in fact, can be fully or partially evaluated. We have developed a technique which allows the transformer to *jump* one or more levels down the metasystem stairway and to let the corresponding machine to make the computation directly, instead of making evaluation in the interpretation mode. We have shown that for this technique to work it is necessary to extend into the negative region the set of possible degrees of metavariables, which by the initial definition could only be positive.

7. We have implemented the technique of metasystem jumps with the *supercompiler* as the program transformer, and the Refal-5 system as the implementation of Refal (the necessary modifications were made in the system). We have shown that metasystem jumps can drastically cut down the computation time, in some cases by a factor greater than twenty. These savings take place when the transformation is carried out according to a three-level MST scheme. In case of schemes of higher orders one can expect even greater savings, which may transform an unfeasible problem into a feasible one.

8. Finally, we have found that it is possible to improve the efficiency of metasystem jumps even further by using an external metacode and making all computing operations dependent solely on the current level relative to a certain *reference level* stored in the machine. Then jumping between

metasystem levels (at least, in the case of a flat functional language) can be achieved by simply changing the reference, without scanning the whole function call, as is done with an internal metacode.

7 Acknowledgements

We are thankful to Dimitri Turchin who made the alterations in the Refal-5 system which were necessary to accomodate for elevated variables. We appreciate the discussion of the ideas presented here at the seminars of the Refal group in Moscow during the summers of 1993 and 1994, especially the remarks by And. Klimov, Ark. Klimov, S. Abramov, S. Romanenko. This paper was reported and discussed in a series of seminars at the University of Copenhagen in July/August 1994. Thanks are due to the participants of the seminars; we appreciate the remarks by N.Jones, M.Sørensen, T.Mogensen, D.Sands. Our special thanks go to R.Glück who read the manuscript and made valuable comments.

References

[1] Abramov, S.M. Metacalculations and logic programming, *Programmirovanie*, **3**, pp.31-44, 1991 (in Russian).

[2] Futamura, Y., Partial evaluation of computation process – an approach to compiler compiler. *Systems, Computers, Controls*, **2**,5 (1971) pp.45-50.

[3] Glück, R., Towards multiple self-application, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (Yale University)*, ACM Press, 1991, pp.309-320.

[4] Glück R., Projections for knowledge based systems, In: Trappl R. (ed.) Cybernetics and System Research, **1**, pp.535-542, World Scientific: Singapore 1992.

[5] Glück R., Jorgensen J., Generating optimizing specializers, In: Internat. Conference on Computer Languages (Toulouse, France) pp.183-194, IEEE Computer Society Press, 1994.

[6] Glück R., Klimov A., Metasystem transition schemes in computer science and mathematics, Accepted for publication in: *World Future:*

the Journal of General Evolution, Gordon and Breach Science Publ., New York.

[7] Jones N., Sestoft P., Sondergaard H., An experiment in partial evaluation: the generation of a compiler generator. In: Jouannaud J.-P. (Ed.) *Rewriting Techniques and Applications*, Dijon, France, LNCS 202, Springer, 1985.

[8] Jones, N. D., Sestoft, P., Søndergaard, Mix: a self-applicable partial evaluator for experiments in compiler generation, in: *Lisp and Symbolic computation* **2**(1), 1989, pp.9-50.

[9] Kistlerov V.L., *Printcipy postroeniya yazyka algebraicheskikh vychislenii FLAC* (The defining principles of the language for algebraic computations FLAC) Institut Problem Upravleniya, Moscow 1987 (in Russian).

[10] Knuth D.E., Morris J.H., Pratt V.R., Fast Pattern Matching in Strings. In: *SIAM Journal of Computer*, **6**(2) pp. 323-350, 1977.

[11] Turchin, V.F., Nirenberg, R.M., Turchin, D.V. Experiments with a supercompiler. In: *ACM Symposium on Lisp and Functional Programming* (1982), ACM, New York, pp. 47-55.

[12] Turchin,V.F. *The Phenomenon of Science*, Columbia University Press, 1977.

[13] Turchin V.F.,Klimov A.V. et al, *Bazisnyi Refal i yego realizatsiya na vychislitel'nykh mashinakh* (Basic Refal and its implementation on computers) GOSSTROY SSSR, TsNIPIASS, Moscow, 1977 (in Russian).

[14] Turchin, V.F. The concept of a supercompiler, *ACM Transactions on Programming Languages and Systems*, **8**, pp.292-325, 1986.

[15] Turchin V., *Refal-5, Programming Guide and Reference Manual*, New England Publishing Co., 1989.

[16] Turchin V.F., Program Transformation with Metasystem Transitions, *J. of Functional Programming*, 3(3) 283-313, 1993.