A Self-Applicable Supercompiler

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

Abstract

A *supercompiler* is a program which can perform a deep transformation of programs using a principle which is similar to *partial evaluation*, and can be referred to as *metacomputation.* Supercompilers that have been in existence up to now (see [12], [13]) were not self-applicable: this is a more difficult problem than self-application of a partial evaluator, because of the more intricate logic of supercompilation. In the present paper we describe the first self-applicable model of a supercompiler and present some tests. Three features distinguish it from the previous models and make self-application possible: (1) The input language is a subset of Refal which we refer to as *flat* Refal. (2) The process of *driving* is performed as a transformation of *pattern-matching graphs*. (3) *Metasystem jumps* are implemented, which allows the supercompiler to avoid interpretation whenever direct computation is possible.

Keywords: program transformation, supercompilation, metacomputation, self-applicaiton, metasystem transition, MST-schemes, metacode, Refal, pattern-matching graphs.

Contact: Valentin F.Turchin, 188 Hiawatha Blvd, Oakland, N.J. 07436.
e-mail Turchin: turcc@cunyvm.cuny.edu
e-mail Nemytykh: nemytykh@scp.botik.yaroslavl.su
e-mail Pinchuk: pinchuk@scp.botik.yaroslavl.su

1 Introduction

Self-applicability of a program transformer is well known to lead to new possibilities, such as automatic creation of compilers from interpreters (see [2], [10], [1], [3]). Self-applicability of a partial evaluator was first achieved in [3]. A *supercompiler* is a program which can perform a deep transformation of programs using a principle which is similar to *partial evaluation*, and is referred to as *metacomputation*. Supercompilers that have been in existence up to now (see [12], [13]) were not self-applicable: this is a more difficult problem than self-application of a partial eveluator, because of the more intricate logic of supercompilation. [1]

In the present paper we describe the first fully self-applicable model of a supercompiler. We concentrate on the three features of this model which distinguish it from the previous models and make self-application possible.

- The input language is simplified to a subset of Refal which we refer to as *flat* Refal (Sec.2).
- The process of *driving* (see [11],[13]), which plays the key role in the construction of the transformed program, is performed as a transformation of *flat pattern-matching graphs* which preserves their functional meaning.
- *Metasystem jumps*, as described in [15], are implemented (Sec.4). This feature allows the supercompiler to avoid interpretation whenever a direct computation is possible.

In Sec.5 the results of the testing of the new supercompiler are discussed.

2 Flat Refal

For the purpose of program transformation, it makes sense to have object programs written in a maximally simplified programming language, into which programs written in a more convenient language can be automatically translated. The supercompiler described here uses *flat Refal* as the language of object programs. Flat Refal is used in two forms different in syntax: the *sentential* form, which is convenient for the human user, and the *pattern-matching graph* form, which is actually transformed in the supercompiler.

[1] Some steps in this direction, though, were reported in [8].

symbol ::= symbolic-name — number — ' character '
expression ::= `[]` — term expression
`[]` ::=
term ::= symbol — variable — (expression)
variable ::= `s`.index — `e`.index
index ::= number — symbolic-name
function-def ::= fn-name { sentences }
fn-name ::= symbolic-name
sentences ::= sentence — sentence sentences
sentence ::= left-side `=` right-side ;
left-side ::= rigid-pattern
right-side ::= expression — function-call
function-call ::= `<` fn-name expression `>`
program ::= sentence — sentence program

Table 1: The syntax of flat Refal

Flat Refal is the lowest level in the hierarchy of the existing versions of Refal which, historically, started from the *basic* version and was then extended up as *extended Refal* and down as *strict* and now *flat* Refal. We wrote the programs constituting the supercompiler in extended Refal. For self-application we first translate the supercompiler from extended Refal into strict Refal; this translation can be fully automatic, but in our tests it was partly manual. Then the text in strict Refal is automatically converted into flat Refal: first in the sentential form (for debugging), and then, finally, in the graph form, in which it becomes an object program for the higher level supercompiler.

Definition of basic, strict and extended Refal can be found in [11], [13], [14]. A brief description of flat Refal follows.

The syntax of flat Refal is given in Table 1. The fundamental data structure of Refal, the *expression*, is more general than the list or s-expression used in Lisp, Prolog, and many functional languages. It allows concatenation as one of the two basic constructions; the other construction is enclosure in parentheses, which makes it possible to represent trees. An expression may be empty, in which case it is represented either by the metasymbol `[]`, or just by an empty space. A string of characters can be represented using only one pair of quotes: `'abc'`.

The two types of variables in Refal correspond to the two basic syntactic types: *s-variables*, such as `s.1` or `s.x`, take exactly one symbol as its value; *e-variables*, as `e.2`, can have any expression as its value.

Definition:
(1) An *object expression* is an expression which includes neither variables, nor function calls.
(2) A *rigid pattern* is an expression such that (a) none of its subexpressions has the form $E_1\mathtt{e}.i_1E_2\mathtt{e}.i_2E_3$, where E_1 etc. are arbitrary expressions, and (b) no e-variable appears in the pattern twice. □

The semantics of Refal is based on pattern-matching. We denote the matching of an object expression E_o, the *argument* of the matching, to a rigid pattern P as $E_o : P$. This is an operation the result of which is either a substitution for the variables in P which transforms P into E_o, in which case matching succeeds, and the substitution is referred to as its *resolution*, or a statement that there is no such substitution (matching fails). A pattern can be seen as a set of object expressions. Therefore, we write $E_o \in P$ if the matching $E_o : P$ is successful.

It can be easily proved (see [11]) that a matching $E : P$, where P is a rigid pattern, has no more than one resolution.

Refal sentences are rewrite rules. The sentences are tried in the order they are written, and the first applicable sentence is used. The right-hand side of every Refal rule in the flat version of the language is either an expression or a single function call; nested function calls are not allowed. (Strict Refal, like flat Refal, requires that the left-hand sides be rigid patterns, but it allows any combinations of expressions and function calls in the right-hand sides. In flat Refal, information exchange takes place only through variables, not through the values taken by function calls. This is an iterative, not recursive, style of programming.

We do not require, though, that the user writes programs in flat Refal; as mentioned above, programs can be written in strict Refal and automatically converted into a flat form. The idea of the translation algorithm is to add to each function one more argument which maintains a representation of the stack of deferred function calls. When the right side of a sentence in the original (not flat) program contains nested calls, the function call which is to be evaluated first is left in the right side of the sentence; all other calls are reworked into *stack elements* and added to the first argument in the required order. If the right side is passive, a special function `Pop` is called, which pops the stack appropriately. We shall not go into details of the algorithm, but only consider, as an example, a simple strict Refal program, and its 'flattening'.

Let function `Fab` be defined by:

```
Fab {
  'a'e.1 = 'b'<Fab e.1>;
  s.2 e.1 = s.2 <Fab e.1>;
    = ; }
```

In a given string of symbols it replaces every letter `'a'` by `'b'`. This program is not in flat Refal, because the recursive right sides are not pure function calls: there is an invisible function which adds `'b'` or `s.2` in front of the recursive call of `Fab`. We treat function `Fab` as if its definition were:

```
Fab {
  'a'e.1 = <Concatenate 'b'<Fab e.1>>;
  s.2 e.1 = <Concatenate s.2 <Fab e.1>>;
   = ; }

Concatenate { e.1 = e.1 }
```

The translation of this definition into the flat Refal is:

```
Fab {
  (e.St) 'a' e.1 = <Fab (e.St (Pop 2)) e.1>;
  (e.St) s.2 e.1 = <Fab (e.St (Pop 3(s.2))) e.1>;
  (e.St) = <Pop (e.St)>; }

Pop {
  (e.St (Pop 2)) e.XXX0 = <Pop (e.St) 'b' e.XXX0 >;
  (e.St (Pop 3(s.2))) e.XXX1 = <Pop (e.St) s.2 e.XXX1 >;
  () e.XXX0  = e.XXX0; }
```

The initial call of `Fab` is now `<Fab () e.1>` (not just `<Fab e.1>`, as before), where the empty content of parentheses stands for the initially empty stack.

If the first symbol of the argument `e.1` starts with `'a'`, the first sentence of `Fab` works. It adds the stack element `(Pop 2)` to the current stack `e.St`. In this element, `Pop` is the name of the function to be called next (formally, this should have been `Concatenate`, but it has an immediate passive outcome, hence `Pop` is called). The symbol `2` is just a case number. Since there are no variables in the deferred function call, no more information is needed than the function name. This is not so in the second sentence, where the element to go to the stack is `(Pop 3(s.2))`. Here `3` is, again, a case number, but the value of the variable `s.2` is also remembered.

When the argument `e.1` becomes empty, function `Pop` is called which starts undoing the stack. The variables of the form `e.XXX`n stand for the value of the latest call. Looking at the first sentence of `Pop` we see that `(Pop 2)` is taken from the stack, then `'b'` added on the left, and the function calls itself recursively. The second sentence works analogously. The stack having been exhausted, the value of the initial function call is returned.

We see that this translation simulates, in an iterative manner, the work of a recursive definition.

¿From the view-point of supercompilation, the difference between strict and flat Refal is not so salient as when we compare the respective programming styles. With strict Refal, configurations of the Refal machine include deferred funciton calls, which are, in the previous supercompilers, represented by a stack. With flat Refal, configurations are flat, but the first argument may represent the same stack with which the strict-Refal supercompiler is working. The advantage of our present approach is that the stack is not a separate structure, but just one of the parameters. It can be treated as other parameters, which is causes considerable simplification. On the other hand, if we want to treat the stack as a separate structure, we still can do it in the flat setting, by giving a special treatment to the first argument.

3 Pattern-matching graphs

Below we briefly describe the most important features of a form of flat Refal programs which we refer to as (flat) pattern-matching graphs. A program in flat Refal is automatically converted into a pattern-matching graph.

In our definition of the pattern matching $E_o : P$ the left side E_o was supposed to be an object expression (no variables). Now we generalize this concept by allowing pattern-matching pairs where the left side E is, an arbitrary expression. The variables in E are supposed to be *bound*, i.e. have definite values (object expressions). The the execution ofgeneralized pattern-matching consists of two steps: first we substitute for the bound variables in E their values, which results in some object expression E_o; then we execute the matching $E_o : P$, where the left side is now an object expression. As a result of a successful matching, the free variables in the pattern P get certain values (object expressions again). If there are no variables in the pattern and the matching is successful, as in `a:a`, the result is denoted as **I**(the identity operation); if matching fails, the result is denoted as $\mathbf{Z}$(impossible operation).

Definition:
(1) A *contraction* is a pattern matching $v : P$, where v is a variable and P is a rigid pattern; we shall denote this contraction as $v \xrightarrow{c} P$.
(2) An *assignment* is a pattern matching $E : v$, where E is an expression and v a variable; we shall denote this assignment as $E \xleftarrow{a} v$.
(3) The *list* of n Refal expressions $E_1, E_2, \ldots, E_n$ is the expression:

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

(4) A *varlist* is a list of free variables where no variable appears twice. One varlist, V_1, is a *subset* of another, V_2, if every variable from V_1 is also in V_2. Two varlists are *equal* (but not necessarily *identical*) if each is a subset of the other. The list of all variables that enter a pattern E is denoted as $\mathbf{var}(E)$.
(5) A *list contraction* is a pattern matching $V : P$, where V is a varlist of n variables and P is a rigid list of the same number of pattern expressions. We shall denote this contraction as $V \xrightarrow{c} P$.
(6) A *list assignment* is a pattern matching $E : V$, where V is a varlist of n variables and E is an arbitrary list of the same number of pattern expressions. We shall denote this assignment as $E \xleftarrow{a} V$. □

We shall often skip the word "list" referring to contractions and assignments when it is clear from the context whether the operation involves one variable or a list of variables.

Our notation of contractions and assignments may seem unusual, but it is quite logical and convenient. It is derived from the following two principles. (1) On the left side we have bound (old, defined) variables; on the right side free (new, to be defined) variables. (2) When the operation is understood as a substitution, the arrow is directed from the variable to its replacement.

To discuss the tests reported in the present paper, we only need to understand contractions and assignments, and have a general idea of the syntax of graphs.

A pattern-matching graph is a tree which represents possible computation paths. We write it in a syntax similar to that of arithmetic expressions. Terms in a sum are walks of the graph which have a common starting node. Concatenation in terms represents sequential execution of the basic operations, while addition (branching) defines various possible cases. Decisions as to which of the paths is to be taken are governed by contractions and *restrictions*, the latter being, essentially, *negative contractions*. Due to the use of restrictions, different branches starting at the same node can be processed independently. This is not so in the case of the sentential form of Refal, where only those cases come to each given sentence which failed at all preceding sentences.

The function call to be computed is given as a graph of the form:

$$\textit{value-list} \stackrel{a}{\leftarrow} \textit{var-list}; \mathbf{fn}(F)$$

where $\mathbf{fn}(F)$ stands for the graph defining the function F. The final result of computation is assigned to the special "external world" variable `e.out`.

4 Metasystem jumps

In self-application of the supercompiler we have a three-level hierarchy of functions: f_0, f_1, f_2 where f_0 is some function, f_1 and f_2 the supercompiler. In this hierarchy a function f_n at the level n is transformed by a function f_{n+1} at the level $n+1$, which is, thereby, a metasystem with respect to f_n. Creation of each new level is a *metasystem transition*, or *MSY* for short.

In [15] a system was developed which makes it possible to automatically change the level at which the computation is done: we call this *metasystem jumps*. Thus, whenever f_{n+1} is to evaluate (partially) f_n, control is first passed to f_n for a direct evaluation, as far as possible. When no step can be made because of unknown values of variables, control is passed back to f_{n+1} for driving. Examples in [15] show that metasystem jumping may result in a speed-up factor of more than 20.

The system developed in [15] was used in the present work. We refer the reader to that paper for details, introducing only the notation used there, since we need it for discussion of tests.

The domain of a function defined in Refal is a set of *object expressions*. Refal programs, however, may use most general Refal expressions, including evaluation brackets and free variables. Hence we cannot directly write Refal programs which manipulate Refal programs. To do this we must map the set of general Refal expressions on the set of object expressions and use the images of "hot" objects, i.e. free variables and evaluation brackets, instead of these objects themselves.

We call this mapping a *metacode*, and denote the metacode transformation of E as $\mu\{E\}$; if E is, syntactically, one Refal term, the curly brackets can be dropped. Obviously, metacoding must have a unique inverse transformation, *demetacoding*, so it must be injective.

E	$\mu\{E\}$
$\texttt{[]}$	$\texttt{[]}$
S	S
$\texttt{s.}i$	$\texttt{('s'}i\texttt{)}$
$\texttt{e.}i$	$\texttt{('e'}i\texttt{)}$
$\texttt{(}E\texttt{)}$	$\texttt{('*'}\mu\{E\}\texttt{)}$
$\texttt{<}E\texttt{>}$	$\texttt{('!'}\mu\{E\}\texttt{)}$
E_1E_2	$\mu\{E_1\}\mu\{E_2\}$

Table 2: The metacode

We used, in the terminology of [15] an *internal* metacode defined by Table 2, where S is any symbol, and i the index of a variable.

In the hierarchy f_0, f_1, f_2 the function f_1 has a metacoded call of f_0 as its argument; f_2 has the metacoded call of f_1 as its argument. We use *MST schemes* for a clear representation of such hierarchies [2]. An MST scheme is built 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:

$$E_1\mu\{E_2\}E_3 \qquad \Longleftrightarrow \qquad \begin{array}{ccc} E_1 & \ldots & E_3 \\ & E_2 & \end{array}$$

The parts of the overall Refal expression which belong to different metasystem levels are put on different lines. Refal 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 an MST scheme into an executable Refal expression, we must metacode each level as many times as long is its distance from the top.

In addition, we can move up some variables in an MST scheme, leaving in the old place a *bullet* $\bullet$. If a variable $t.i$ (type t, index i) is raised by h levels – the number referred to as its *elevation* — then the meaning of the bullet-variable pair is $\mu^{n-h}\{\texttt{<Dn}^h t.i\texttt{>}\}$, where $\texttt{Dn}$ (read 'Down') is a Refal function which performs the metacode transformation on object expressions. Thus a bullet-variable pair can be seen as a variable metacoded $n-h$ times, but the

[2]This notation was first used by one of the present authors (VT) in lectures at the 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 [5].

possible values of this variable are only such that they an object expression h times. In particular, if an input variable is of an elevation h, its desired value must be metacoded h times before the beginning of computation. (See more about elevated variables in [15]).

Moving variables up radically changes the way the expression represented by the scheme will be executed. To read an MST scheme correctly, the following *rule of two levels* can be used:

The variables on the top level are free. They are the arguments of the function represented by the MST scheme as a whole; some specific values must be substituted for them before computation. The variables on the next level down are the variables of the function whose definition (the program for it) results from the computation on the top level.

As an example, consider the well-known problem of compilation. Let `<L program,data>` be a Refal interpreter of some language L. Let P be a program in L. Using the supercompiler we can translate P into an efficient program in Refal by partial evaluation according to the MST scheme:

```
<Scp ................>
    <L P ,e.data>
```

Now suppose we want a *function* which would accept any arbitrary program, not just P. If we simply put the variable `program` instead of P:

```
<Scp ......................>
    <L e.program, e.data>
```

we will not get what we want. Here the variables for data and for program are on the same level and are treated in the same way. Using the rule of two levels we see that the result will be a function of two variables: this is, again, an interpreter. No partial evaluation takes place, because the value of `e.program` remains unknown. If we want partial evaluation, we must raise the variable `e.program` to the level of `Scp`:

```
<Scp ...e.program.........>
    <L    •    , e.data>
```

Now the rule of two levels tells us that a specific program will be asked by this function as the value of the argument `e.program`. Then partial evaluation will take place, returning a program for a function of the variable `e.data`. It is, of course, a translation of `e.program`. The function defined by this MST scheme may further be subject to a transformation by another partial evaluator. It will then produce an efficient compiler for L.

It should be noted that in the calls of function transformers such as `Scp` there is, implicitly, one more argument, which is not shown in our MST schemes: the definition (program for) all function calls on the object level.

5 Testing

Our system is still under debugging, but we can, already, show a few interesting tests. We have tested our supercompiler on several "classical" for partial evaluation problems, such as string pattern-matching and compilation, but here we are concerned only with self-application and similar problems.

5.1 Test 1

Before attempting self-application, we ran a few tests which demonstrate "almost self-application": a three-level scheme where `Scp` transforms a metainterpreter applied to some program. One of the simplest tests is:

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

Here `Fab` is a simple function which changes every `'a'` to `'b'`. Its definition in flat Refal is:

```
Fab { e.1 = <Fab1 () e.1> }
Fab1 {
   (e.1) 'a'e.2 = <Fab1 (e.1'b') e.2>;
   (e.1) s.3 e.2 = <Fab1 (e.1 s.3) e.2>;
   (e.1) = e.1; }
```

Function `Int` is an interpreter of flat Refal. The argument x of `Fab` is *elevated* by one level to become free for `Int` and bound for `Scp`. The call of `Int`, by its definition, computes `<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 program, which is, as it should be expected, an exact copy of the original program for `Fab`, except for the names of variables.

5.2 Test2

The supercompiler `Scp` at the top level of the MST-schemes in self-application is denoted as `Scp2`. It was written in extended Refal; its volume: 302 sentences. It was manually translated into strict Refal; the volume became 347 sentences. Then it was automatically translated into a flat form (499 sentences), which then was used as `Scp1`, to which `Scp` was applied. Thus, `Scp1` and `Scp2` represent the same algorithm written in two versions of Refal. We could use `Scp1` in the place of `Scp2` to have "very strict" self-application, but we used `Scp2` for the convenience of testing.

In Test 2 the scheme of self-application was:

```
<Scp2 ............. >
    <Scp1 ....... >
        <Fab e.1>
```

This is a correctness check. All work is done by `Scp1`, with `Scp2` as a simple supervisor. We see from the rule of two levels that `Scp1` is a function of no variables (a constant) whose value is the definition of a function of `e.1` which is equivalent to `Fab`. The supercompiler on the level 2, `Scp2`, has no free variables on either top, or the next level; it transforms a function (namely, `Scp1`), which is a constant. The output of `Scp2` is not the same as the output of `Scp1`, because they are on different metasystem levels. If `Scp1` is the constant *const*, then `Scp2` outputs the metacode of a program (in the graph form) which outputs *const*, i.e. $\mu\{const \stackrel{a}{\leftarrow} \texttt{e.out}\}$.

The actual output of the supercompiler is, however, a bit more complex. The program produced by the supercompiler can be seen as the expression:

$$(\texttt{Define}\ def(f_1)\ldots def(f_n))(\texttt{Call}\ G_{in})$$

which combines a list of the definitions of new functions f_i and the initial function call to compute G_{in}. The function definition is of the form:

$$def(f_i) = mu\texttt{<}Config\texttt{>} = \texttt{Fn}i : \mu\{Prog_i\}$$

where $< Config >$ is a recurrent configuration in terms of the initial program, $\texttt{Fn}i$ is the symbolic name of the corresponding new function f_i, and $Prog_i$ the program for it.

The output of the 2nd-level supercompiler `Scp2` is a metacoded program which we shall denote as $\mu\{Prog2\}$. In Test 2 it is this:

$$\mu\{Prog2\}\ \texttt{= (Define) (Call}\ \mu\{\mu\{Prog1\} \stackrel{a}{\leftarrow} \texttt{e.out}\})$$

This program simply outputs the metacoded program $Prog1$:

$$\mu\{Prog1\}\ \texttt{= (Define}\ \mu\texttt{<Fab1(e.23)e.24>=Fn1:}\mu\{Prog0\}\texttt{)}$$
$$\texttt{(Call}\ \mu\{\texttt{()(e.1)}\stackrel{a}{\leftarrow}\texttt{(e.23)(e.24)}\mathbf{fn}\texttt{(Fn1)}\}\texttt{)}$$

Program $Prog1$ is the result of supercompilation by Scp1 of the call <Fab e.1>; therefore it must be equivalent to it. $Prog1$ defines one recursive function Fn1 which is the supercompiled function call <Fab1(e.23)e.24>. The variables of this configuration become the variables of the definition of $Prog0$. Therefore, the initial call is a call of Fn1 after the necessary assignments to the variables (e.23)(e.24). The program $Prog0$ is the program for Fn1; it results from supercompilation of Fab1, and it is identical to it, except for the different names of variables. This is exactly what one should have expected.

5.3 Test 3

The MST scheme in this test is:

```
<Scp2 ................. >
    <Scp1 ... s.2 .... >
        <Fab    •   e.1 >
```

Here the argument of Fab is s.2 e.1, i.e. it starts with some symbol s.2. We have raised s.2 to become a free variable in the call of Scp1. According to the Rule of two levels, the result of Scp2 must be a function of s.2. The actual output is:

$$Prog2 = \texttt{(Define)(Call } \mu\{Prog1\}\texttt{)}$$

where $Prog_1$ is the initial graph, which we present in the sentential form for readability:

```
Prog1 {
   'a' =  (Define μ<Fab1 ('b')e.1>=Fn1: μProg01)
          (Call μ{()(e.1)←a(e.23)(e.24); fn(Fn1)});
   s.2 =  (Define μ<Fab1 (s.2)e.1>=Fn2: μProg02)
          (Call μ{()(e.1)←a(e.23)(e.24); fn(Fn2)});
    }
```

We see that Prog1, indeed, depends on s.2 and includes two cases where its value is 'a', or any value distinct from 'a'. In both cases the structure of the value is similar to that in the preceding test. The difference is that *partial evaluation* took place with respect to s.2. Look at the recurrent calls of Fab1: it is already within the parentheses which enclose the ready part of the string, changed to 'b' if it was 'a'. The programs $Prog01$ and $Prog02$ for Fn1 and Fn2, respectively, are:

```
Fn1 {
   (e.23) 'a'e.24 = <Fn1 (e.23'b') e.24>;
   (e.23) s.25 e.24 = <Fn1 (e.23 s.25) e.24>;
   (e.23) = 'b'e.23; }

Fn2 {
   (e.37) 'a'e.38 = <Fn2 (e.37'b') e.38>;
   (e.37) s.39 e.38 = <Fn2 (e.37 s.39) e.38>;
   (e.37) = μ^-1{s.2} e.37; }
```

These programs are the same as the initial program for `Fab1`, except that they are modified because of the partial evaluation which took place. Functions `Fn1` and `Fn2` depend on `e.1` alone; `s.2` has already been processed and is kept in the first argument of `Fab1`. When the string is exhausted and the output is done, `s.2` must be added at the beginning of the string resulting from processing `e.1`, converted to `'b'` in `Fn1`, or left unchanged in `Fn2`.

The variable `s.2` enters the program P_2 in the *negative* degree of metacode. This is the result of the variable `s.2` being *elevated*. Recall that the meaning of a variable elevated by h levels and metacoded $n-1$ times is $\mu^{n-h}\{¡\mathrm{Dn}^h s.2¿\}$. In our case $h = 1$, and $n = 2$. When $\mu\{\mathrm{Prog1}\}$ which is yielded by $Prog2$ is demetacoded, the metacode level from 1 becomes 0, but the elevation h remains 1, which means that the value of `s.2` is a metacoded (once) symbol. When it is substituted by $Prog1$ in $\mu\{Prog01\}$ and $\mu\{Prog02\}$, we have a correct – all on the level 1 – program. When it is demetacoded as a whole the result will be a correct working program (level 0); the value of $s.2$ will be demetacoded together with the rest of the program. However, if we want to write down the demetacoded program for `Fn2` for an arbitrary `s.2`, we must insert in the proper place the demetacoded symbol which is the value of `s.2`. This is exactly what $\mu^{-1}\{\mathrm{s.2}\}$ means.

It should be noted that with our metacode, symbols do not change under metacoding or demetacoding. Yet it would be an error to put `s.2` instead of $\mu^{-1}\{\mathrm{s.2}\}$, because `s.2` is not a free variable in the definition of `Fn2`. The sentence which results from such a replacement would be syntactically faulty: the right side of the sentence would include a variable which does not enter the left side.

5.4 Test 4

The MST scheme is:

```
<Scp2 ...............>
    <Scp1 ... e.1...>
        <Fab ..●..>
```

Here we have elevated the whole argument `e.1`. Because of this, `Scp1` is *not* transforming the program for `Fab`. It has no bound variables, therefore the call of `Fab` is a *constant* for it. What is going on is this. `Scp1` gets a value E of `e.1` at input, computes the value E' =<`Fab` E>, and then outputs a program which outputs the constant E'. As for `Scp2`, it outputs a program which is equivalent to `Scp1`, i.e. it does the same thing, but much more efficiently; ideally, as efficient as the function `Fab` itself outputs is value.

The actual result of supercompilation is:

$$Prog2 = \texttt{(Define } \mu\texttt{<}Conf\texttt{>=Fn1:} \quad \mu\{Prog1\}\texttt{)}$$
$$\texttt{(Call } \mu\{\texttt{()(e.1)} \stackrel{a}{\leftarrow} \texttt{(e.59)(e.60); } \mathbf{fn}\texttt{(Fn1)}\}\texttt{)}$$

Here $Conf$ is one of the configurations in the supercompilation of `Scp1` by `Scp2`. It is big and hardly manageable by a human user. When `Scp2` works on `Scp1` it finds recurrent configurations, which become new recursive functions. $Conf$ is such (and the only) configuration. The only thing about it that is easy to establish is that it includes two free variables: `e.59` and `e.60`.

The program $Prog1$ is:

$$
\begin{array}{l}
\texttt{Fn1 } \{ \\
\quad \texttt{(e.59) 'a'e.60 = <Fn1 (e.59'b') e.60>;} \\
\quad \texttt{(e.59) s.72 e.60 = <Fn1 (e.59 s.72) e.60>;} \\
\quad \texttt{(e.59) = (Define)(Call e.59 } \stackrel{a}{\leftarrow} \mu\{\texttt{e.out}\}\texttt{);} \\
\quad \texttt{e.59 = (Define) (Call } \mathbf{Z}\texttt{); } \}
\end{array}
$$

Since the variable `e.1` has the elevation 1, its intended value must be metacoded before the computation by `Scp1`. This remains true also for `Scp2`, because it produces a program which faithfully follows the input-output requirements for `Scp1`.

At the first glance it may seem that the assignment in the third sentence of `Fn1` has an error: the output variable `e.out` is metacoded, while `e.59` is not. But `e.59` is, like `e.1` elevated; indeed, it is the result of processing `e.60`, and the initial value assigned to `e.60` is `e.1`. Function `Fn1` outputs a metacoded program. When it is demetacoded, $\mu\{$`e.out`$\}$ becomes `e.out`, and *the value of* `e.59` is demetacoded. So, the result is as it should be.

Function `Fn1` mimicks the recursive function `Fab1`, except that it has one more sentence: the fourth. This also is a consequence of the fact that

`Scp1` works in metacode. `Fab1` is defined in expectation that its second subargument is always a string of symbols. If it starts with a parenthesis, the Refal machine fails (comes to an abnormal stop); the value of the function in this case is not defined. But when `Scp1` works in metacode and discovers a parenthesis it *does not fail*. In our language of graphs there is a special operation for this situation: **Z**. This is our way to say in metacode that the original function causes failure. So, if none of the first three sentences of `Fn1` works, then the fourth sentence assigns **Z** as the output value.

It is interesting to compare Tests 3 and 4 with regard to the level at which recursion loops are closing. The `Define` part of the output consists of those configurations which were found recurrent. In Test 3 the function transformed by `Scp2` depends on a symbol variable only; there is no recursion on this argument (see the MST scheme). Therefore, the `Define` part of the output is empty. The function transformed by `Scp1`, on the contrary, depends on an e-variable, thus on this level we have two recurrent configurations `Fn1` and `Fn2`, in the `Define` part. In Test 4 `Scp2` works on a recursive function of an e-variable, hence the appearance of recurrent configurations is inevitable.

5.5 Test 5

The next two tests demonstrate another use of a supercompiler: in a combination with a non-standard interpreters of the object language. Our language, flat Refal, does not allow us to make lazy evaluation. To amend this, we have written in flat Refal a *lazy* interpreter `Lazy-Int` of a full functional language (strict Refal). Given a program in strict Refal, we interpret it by `Lazy-Int` and supercompile this process by `Scp` according to the following MST-scheme:

```
<Scp ..................... >
   <Lazy-Int .. (e.1)... >
        <Fbc <Fab • >>
```

The program which is implicit in the call of `Lazy-Int` defines two functions: `Fab` and `Fbc`. Function `Fab` was defined in Sec. 2 as an example of a non-flat recursive function:

```
Fab {
  'a'e.1 = 'b'<Fab e.1>;
  s.2 e.1 = s.2 <Fab e.1>;
    = ; }
```

It replaces every `'a'` by `'b'`. Function `Fbc` is defined analogously and replaces every `'b'` by `'c'`. The lazy interpreter works on the composition of these two functions. The expectation was that after supercompilation we should have a flat one-loop program which converts both `'a'` and `'b'` into `'c'`.

And indeed, the resulting program is exactly as expected:

```
Fac {
    () = ;
    (e.1 ) = <F1C1 ()(e.1 )> ; }

F1C1  {
    (e.25 )('a'e.26 ) = <F1C1 (e.25 'c')(e.26 )> ;
    (e.25 )('b'e.26 ) = <F1C1 (e.25 'c')(e.26 )> ;
    (e.25 )(s.29 e.26 ) = <F1C1 (e.25 s.29 )(e.26 )> ;
    (e.25 )() = e.25 ; }
```

It transforms a two-pass program into an equivalent one-pass program.

5.6 Test 6

In this test we use the supercompiler in a combination with an even less standard interpreter: an inverse interpreter `Inv-Int` which executes a function definition (in a form of a graph) from its ends to its beginnings, and computes the value of the input when an output is given. `Inv-Int` stops after finding the first solution and is relatively simple, because we have put rather strong constraints on its input – a program to invert. They are:

- The program must be in flat Refal.
- The right side of each sentence, or the argument of a function call in the right side, must be a rigid pattern.
- All variables of the left side of a sentence must be met also in the right side.

The MST-scheme in Test 6 was:

```
<Scp ............................. >
    <Inv-Int .......... (e.1)... >
                <Rev e.x >
```

The interpreter `Inv-Int` expects a configuration which depends on the input variable e.x, and the output value assigned to the free variable `e.1`. Implicit in the call is the program which defines the function `Rev`:

```
Rev { e.x = <Rev1 (e.x) ()>; }

Rev1 { (s.1 e.x) (e.y) = <Rev1 (e.x) (s.1 e.y)>;
() (e.y) =  e.Y; }
```

The interpreter finds a value of `e.x` such that

$$\texttt{<Rev e.x>} = \texttt{e.1}$$

and gives it out as its value.

Function `Rev` reverses its argument. Since its reversal is itself, the best we can expect from the supercompiler is that it returns a function definition identical to the above program for `Rev`. And it does so.

We can see this result as a proof that $\texttt{Rev}^{-1} = \texttt{Rev}$, and therefore, a proof that `<Rev<Rev<e.1>>` is the identical function. It shoud be noted that this double reverse configuration cannot be reduced by direct supercompilation to the identical function – even such a function that passes through the string symbol by symbol, without changing them.

5.7 Conclusion

What we have shown in these tests is the hard core of the supercompiler system. For specific applications, special front and back ends can be added. For instance, the system requires that the value of an elevated variable be metacoded at input and demetacoded at output. Clearly, this can be included into the system, and the end user need not know anything about this. MST-schemes can also be formed for a wide class of problems without bothering the user with such details.

Our tests show that a supercompiler can be self-applicable. The important thing here is not, of course, just to apply it to itself and obtain *some* target program; this program must be a good, efficient program. We gave examples of such non-trivial transformations. A big program (`Scp`) is applied to a big program (`Scp` itself or a non-standard interpreter) and produces a small residue, which is the desired solution to the problem. Sure enough, the zero-level programs in these tests are still small, not big, but we believe that this is a matter of some technical details: eliminating bugs and fine-tuning some of the algortihms, such as generalization. Writing supercompilers, like writing compilers, is a *technology*. It develops slowly but surely, by accumulating small discoveries and improvements.

Acknowlegements. First tests of the present supercompiler were discussed in the summer of 1994 in Copenhagen, Moscow and Pereslavl (Russia). The authors thanks the participants – especially Sergei Abramov, Sergei Chmutov, Robert Glück, Neil Jones, Andrei Klimov, Arkadi Klimov, Torben Mogensen, Sergei Romanenko, Morten Sørensen – for useful comments.

References

[1] Ershov, A.P. On the essence of compilation, In: E.J.Neuhold(ed) *Formal Description of Programming Concepts*, pp.391-420, North-Holland, 1978.

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

[3] 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.

[4] Jones, Neil. The essence of program transformation by partial evaluation and driving. In: *Proc. of The Atlantique Worlshop on Semantics Based Program Manipulation*, N.Jones and C.Talcott Ed. Copenhagen University, pp.134-147, 1994.

[5] 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.

[6] Glück, R. and Klimov, And., Occam's razor in metacomputation: the notion of a perfect process tree. In: *Static Analysis*, COusot et.al (Eds), LNCS Vol 724, pp.112-123, Springer-Verlag 1993.

[7] Glück R. and Sørensen, M.H. Partial deduction and driving are equivalent. In: *Symposium on Programming Language Implementation and Logic Programming (PLILP'94)*, LNCS, Springer-Verlag, 1994.

[8] Glück R. and Turchin V., Experiments with a Self-applicable Supercompiler, CCNY Technical Report, 1989.

[9] Turchin, V.F., Equivalent transformations of recursive functions defined in Refal. In: Teoriya Yazykov I Metody Postroeniya Sistem Program-

mirovaniya (Proceedings of the Symposium), Kiev-Alushta (USSR), pp.31-42, 1972 (in Russian).

[10] 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).

[11] Turchin, V.F. *The Language Refal, the Theory of Compilation and Metasystem Analysis*, Courant Computer Science Report #20, New York University, 1980.

[12] 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.

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

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

[15] Turchin V., Nemytykh, A. Metavariables: Their implementation and use in Program Transformation, CCNY Technical Report CSc TR-95-012, 1995.