Metacomputation: MST plus SCP

Valentin F. Turchin
The City College of New York

First of all, I want to thank the organizers of this seminar for inviting me to review the history and the present state of the work on supercompilation and metasystem transitions. I believe that these two notions should be of primary importance for the seminar, because they indicate the lines of further development and generalization of the two key notions most familiar to the participants: supercompilation is a development and generalization of partial evaluation, while metasystem transition is the same for self-application. For myself, however, the order of appearance of these keywords was opposite: I started from the general concept of metasystem transition (MST for short), and my consequent work in computer science has been an application and concretization of this basic idea.

1 History

Consider a system S of any kind. Suppose that there is a way to make some number of copies of it, possibly with variations. Suppose that these systems are united into a new system S' which has the systems of the S type as its subsystems, and includes also an additional mechanism which somehow examines, controls, modifies and reproduces the S-subsystems. Then we call S' a *metasystem* with respect to S, and the creation of S' a *metasystem transition*. As a result of consecutive metasystem transitions a multilevel hierarchy of control arises, which exhibitss complicated forms of behavior.

In my book *The Phenomenon of Science: a Cybernetic Approach to Evolution*, [38] I have interpreted the major steps in biological and cultural evolution, including the emergence of the thinking human being, as nothing else but metasystem transitions on a large scale. Even though my Ph.D. was in theoretical physics, I was so excited about my new cybernetic ideas that I shifted from physics to computer science and left the atomic center in Obninsk for the Institute for Applied Mathematics in Moscow.

An extra-scientific factor came into play in the late 1960s: I became an active member of the human rights movement. My book was written about 1970, but it could not be published in the Soviet Union solely because of the author's name. It took seven years to smuggle it to the West and have it published in the English language.

The first step towards MST in computers was to design an appropriate algorithmic language. I called the first version of such a language *meta-algorithmic*, [35], since it was supposed to serve as a metalanguage for defining semantics of algoritmic languages. Then I simpilfied it into what was named REFAL (for REcursive Functions Algorithmic Language). Refal was conceived as the universal language of metasystem hierarchies, which is, on one hand, simple enough, so that the machine that executes programs in this language (the *Refal machine*) could become an object of theoretical analysis — and, on the other hand, is rich enough to serve as a programming language for writing real-life algorithms, unlike such purely theoretical languages as the language of the Turing machine or Markov's normal algorithms (the latter, by the way, was one of the sources of Refal). Even to this day Refal remains, in my (biassed, no doubt) view the most successful compromise between these two requirements.

An efficient interpreter for Refal was first made in 1968 [25]. At that time Refal was very different from other languages, being a purely functional language with built-in pattern-matching. Now, after quite a few functional languages similar to Refal have appeared, I need only to summarize its distinctive features, in order to make it familiar.

The most distinctive feature of Refal is its data domain. Unlike other functional languages which use *lists* (skewed binary trees) [1] for symbol manipulation, Refal uses *expressions*, which can be seen as trees with arbitrary (unfixed) arity of nodes, and are defined formally as:

$$term ::= symbol \mid (expression) \mid \texttt{<}function\ expression\texttt{>}$$
$$expression ::= empty \mid term\ expression$$

Variables in Refal have their intrinsic types shown by a prefix; thus `s.`i, where i is the index (name) stands for an arbitrary symbol, `t.`i an arbitrary term, and `e.`i an arbitrary expression, e.g. `s.1, t.2, e.x`. In the case of an expression variable, the prefix may be dropped: `x` is the same as `e.x`.

We use angular brackets to form function calls: `<`$f\ x$`>`. A program in Refal is a sequence of *sentences* (replacement rules) which are tried in the

[1] They first appeared in Lisp, so I often refer to lists as Lisp's data domain

order they are written. Here is an example of a function `f` which traverses its argument from left to right and replaces every `'a'` by `'b'`:

```
<f 'a'> = 'b'<f x>
<f s.1 x> = s.1 <f x>
<f > =
```

By the year 1970 I was lucky to have gathered a circle of young people, the Refal group, which met regularly, survived my emigration and is still alive and well. As in any informal group, some people went, new people came. I am deeply thankful to all of them, and especially to those most active and persistent: Sergei Romanenko (the first to have come), Nikolai Kondratiev, Elena Travkina, Andrei Klimov, Arkadi Klimov, Viktor Kistlerov, Igor Shchenkov, Sergei Abramov, Alexei Vedenov, Ruten Gurin, Leonid Provorov, Andrei Nemytykh, Vika Pinchuk. I will always remember Alexander Romanenko who died before his time in 1993.

Together with a smaller group at the Moscow Engineering and Physics Institute (Stanislav Florentsev, Alexander Krasovsky, Vladimir Khoroshevsky) we made Refal compilers for the most popular machines in the Soviet Union, and Refal became pretty well known in that part of the world. It is not my intention to focus on Refal (a bibliography on the development and use of Refal compiled a few years ago includes about 200 items), but I want to mention two further outgrowths of Refal: The language FLAC for algebraic manipulation developed by V.Kistlerov [21], [4], and Refal Plus by S.Romanenko and R.Gurin [16], which is a kind of logical closure of the ideas on which Refal is based. In may later work I used the extended version of Refal named Refal-5, [43], which is operational under DOS and UNIX.

The next conceptual station after fixing the language was *driving* [36], [37]. Suppose we have the call `<f 'a'x>` of the function `f` above. Obviously, it can be replaced by `'b'<f x>`, because this is what the Refal machine will do in one step of function evaluation. *Partial evaluation* took place, which is a simple case of driving. Now take the call `<f x>` where partial evaluator has nothing to do. Driving, however, is still possible, because it is simulation of one step of computation in any circumstances. When you see equivalent transformation of programs as the use of some equations, our partial evaluation is the use of the equation `<f'a'x>` =`'b'<f x>`, and this makes perfect sense. But, given the call `<f x>`, there is no equation in sight which would improve the program. Driving is a product of cybernetic thinking. We create a metamachine, and the first thing it must be able to do with the Refal machine is to simulate its behavior. Thus, the metamachine

drives the Refal machine, forcing it to do somehting for which it was not originally prepared: make computation over expressions with free variables. In our case driving will produce the graph:

```
[1]<f x>  x:'a'x; [2]'b'<f x>
        + x:s.1:x; [3] s.1 <f x>
        + x:[]; []
```

where the original call is the root node labeled by `[1]`, and there are three edges, separated by `+`, which lead to three resulting expressions, according to the three sentences in the definition of `f`. By $e : p$ we denote the operation of matching expression e to pattern p; a pattern is a special kind of an expression (referred to as *rigid*), which guarantees the uniqueness of the operation. `[]` stands for an empty expression for readability.

¿From driving I came to supercompilation (SCP for short). Let me briefly describe this technique of program transformation leaving aside many details and variations.

A supercompiler is an upgrade of a driver. When a new *active* (i.e. representing a function call) node C appears in driving, the supercompiler examines its ancestors and with regard to each ancestor C' takes one of the three decisions:

1. Reduce C to C' with a substitution; this can be done only if $C \subseteq C'$.
2. Generalize C and C', i.e. find such C^g that $C \subseteq C^g$ and $C' \subseteq C^g$; then erase the driving subtree which starts at C', reduce C' to C^g and go on driving C^g.
3. Do nothing on C' (the nodes C and C' are "far from each other"), and examine the next ancestor. If there are no more ancestors, go on driving C.

Supercompilation ends when the graph becomes self-sufficient, i.e. for every active node shows how to make a transition to the next node which corresponds to at least one step of the Refal machine. This graph is a program for computation of the function call represented by the initial node.

Note the difference between the principle of supercompilation and the usual idea of program transformation, where the program is changed step by step by applying some equivalences. In supercompilation we never change the original program. We regard it as a sort of "laws of nature" and construct a model of a computation process governed by these laws. When the

model becomes self-sufficient, we simply throw away the unchanged original program.

When we discussed supercompilation in the seminars of the Refal group, I always stressed my belief that metasystem transition is important in itself. If it has been the key at all stages of biological and technical evolution, how can we hope to create really intelligent machines without the use of this principle? We should develop techniques of dealing with repeated metasystem transitions; applications are bound to emerge.

The first confirmation of this belief came when I figured out that supercompilation of an interpreter is compilation, and supercompilation of the supercompiler that does compilation (second MST) yields a compiled (efficient) compiler. Moreover, making the third MST we can obtain a compiled compiler generator. I was very excited about my discovery, and told Andrei Ershov about it. Ershov at that time worked on partial evaluation – the first MST – but he did not know about the second MST, and he also got excited[2]. Neither he, nor I knew at that time that Yoshi Futamura [8] made this discovery a few years before me. Andrei saw a reference to Yoshi's paper, but could not get the journal. In his big paper [6] he referred to my result as "Turchin's theorem of double driving".

That was in 1976. Since 1974 I had been jobless, home-searched and periodically interrogated by the KGB. Meanwile a book on Refal and its implementation was written by several members of the Refal group, including myself, [3]. My friends managed to get permission for publishing it as a technical material of the institue where I worked before being fired – on the condition that I will not be in the list of authors. To meet this requirement they decided not to include the list of authors at all. The book was published anonymously. But I smuggled into it a page or so about my results on automatic production of compilers.

I emigrated in 1977, and it took some time to take roots in the new environment: a process which can never be fully successful. For a year and a half I stayed at the Courant Institute of NYU and used this time to write a 250 pages report [39], where I summarized the ideas and results of the past years and also sketched some new ideas to turn to them later. Then I got a position at the City College of the City University of New York.

I wrote the first supercompiler (SCP-1) in 1981-82 [40] with the help of Bob Nirenberg and my son Dimitri in carrying over the implementation of Refal from Russia to the US and upgrading it. ¿From the examples in

[2]Ershov describes our meeting in detail in [7]

[40] one could see that supercompilation includes, but is much stronger as a transformation technique than partial evaluation (which takes place automatically in driving). The examples in [40] included program specializaiton, but when I tried self-application (2nd MST), I met a dozen of technical difficulties.

Partial evaluation is simpler then supercompilation, so automatic generation of a compiler from an interpreter was first achieved by self-application of a partial evaluater (2nd and 3rd MST). This was done by Neil Jones and co-workers at DIKU, Copenhagen, [17], [26], [18], [19], and was an important step forward and a great success. Self-applicable partial evaluation became an established field of research. Of the members of the Moscow Refal group, Sergei Romanenko and Andrei Klimov contributed to this field, [22], [29], [30], but I decided to concentrate on further development of techniques of supercompilation, and wrote a stronger an better supercompiler, SCP-2.

The fall semester of 1985 I spent at DIKU in Copenhagen invited by Neil Jones. This was a very important visit for me. I got a chance to discuss in detail my ideas on MST and SCP with Neil and other people at DIKU. Among other things, these discussions helped me to finalize and have published my paper on supercompilation [41]. Since then I visited DIKU on several occasions and always had useful discussions and enjoyed the friendly atmosphere there.

In 1988-89 Robert Glück, then at the Vienna Technical University, joined the MST-SCP Project. He spent a year and a half at the City College working with me on its various aspects. We managed to achieve self-application of SCP-2 in some simple cases, but it became clear that a thorough overhaul of the supercompiler is needed. After returning to Europe Robert went to Copenhagen. He carried over the techniques of metacoding and doing MSTs, which was originally developed for the Refal supercompiler, to partial evaluation and list-based languages. In the early work on partial evaluation by N.Jones with co-workers [19] it was stated that in order to achieve good results in self-application, a preliminary binding time analysis ("off-line") was necessary. However, Robert showed ([11]) that with a correct use of metacoding, partial evaluation is self-applicable without any binding time analysis: "on line".

Little by little, more people got involved in the work on MST-SCP ideas, including some members of the Moscow Refal group. Andrei Klimov started active work in this field and found a mate in Robert Glück [13, 14, 15]. Alexander Romanenko started work on function inversion [27, 28]. Sergei Abramov did an excellent and potentially very important work, [1] on pro-

gram testing on the basis of *driving with a neighborhood* (see Sec.3.3), and wrote a monograph on metacomputation other than supercompilation, [2].

Neil Jones [20] gave a thorough theoretical analysis of driving as compared to partial evaluation. More papers on supercompilation have appeared recently from DIKU, [12, 31, 32, 33, 34]. The role of Robert Glük in that, as one can see from the references, has been of primary importance.

Morten Sørensen wrote a Master thesis on supercompilaiton, [31]. Its title, *Turchin's Supercompiler Revisited*, which pleased me very much because it symbolized the emerging interest to my work. Then he made a suggestion that the Higman-Kruskal theorem on homeomorphic embedding be used as the criterion for generalization in supercompilation, [32], which probably will be judged as one of the most important contributions to the SCP techniques during the last few years; I discuss it in more detail in Sec.2.2.

In 1993 I decided to restrict Refal as the object language of SCP to its *flat* subset where the right side of a sentence cannot inlcude nested function calls: it is either a passive expression, or a single function call. The purpose, of course, was to simplify the supercomiler, for self-application to become possible. I started writing such a supercompiler, SCP-3. In September 1993 Andrei Nemytykh from the Programming Systems Institute (Perslavl, Russia), came to CCNY for an academic year under a grant from the National Research Council. Working together, we have made SCP-3, at long last, self-applicable. That was in the spring of 1994, [45, 46]. Returning to Russia, Andrei continued the work on SCP-3 with Vika Pinchuk. I do not speak more on this because SCP-3 with some examples of its performance is presented in a separate paper at this symposium.

In the summer of 1993 Yoshi Futamura invited me to spend a month in Tokyo to discuss supercompilation in relation to his concept of *generalized partial computation*, [9, 10]. There are common aspects, indeed. In both approaches the information about the states of a computing machine goes beyond listing the values of some variables. The main difference is that Yoshi relies on some unspecified theorem proving, while my point is to do everthing by supercompilation, including theorem proving.

In July 1995, invited by José Meseguer, I spent a pleasant week at Stanford Research Institute in California explaining and discussing the details of SCP and MST. José and his graduate student Manuel Clavel are working on reflective logics and languages in the frame of Meseguer's theory of general logics [23], [24]. Their goal is to extend the techniques of supercompilation to their systems in order to improve their efficiency, and I know that Manuel has already made some progress along this path.

My review of events and ideas concerning MST and SCP is, no doubt, incomplete, and so is my bibliography. I ask for forgiveness in advance.

In Sec.2 I discuss a few important aspects of supercompilation. In Sec.3 I give an outline of a few ideas which have not yet been properly translated into computer programs. My exposition is very informal and sketchy. I try to have it done through a few simple examples. The field of MST+SCP has its own formalism, which I, obviously, cannot systematically present here. Yet I hope that ther reader unfamiliar with it will still be able to figure out what it all is about, without being going into formal details.

A word on terminology. In 1987 I suggested *metacomputation* as an umbrella term covering all computation which includes at least one metasystem transition. By this definition, partial evaluation is also a variety of metacomputation, and the term could be used for identifying such meetings as this seminar. However, partial evaluation people have been in no hurry to use it. In contrast, it is readily used by people in the field I denote in this paper as MST plus SCP (sounds barbarian, of course). Thus, for the time being, at least,

$$metacomputation \approx MST + SCP$$

2 Aspects of supercompilation

2.1 Pattern-matching Graphs

I will reveal a small secret. Our supercompilers do not actually use Refal as the language of object programs; they use the language of *pattern-matching graphs*, also referred to as *Refal graphs*. The program in Refal to be supercompiled is first automatically translated into the graph form, and the output of the SCP is also a Refal graph.

Algebraically, a pattern-matching graph is a sum of products of three varieties of the operation of pattern-matching, with a product implying sequential, and a sum parallel, execution. 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 $v \stackrel{c}{\rightarrow} p$. An *assignment* is a pattern matching $e : v$, where e is an expression and v a variable; we shall denote this assignment as $e \stackrel{a}{\leftarrow} v$.

This notation may seem unusual (especially that of an assignment), but it is logical and quite 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.

Examples. $\texttt{x}\overset{c}{\rightarrow}\texttt{s.1 x`a'}$ is a contraction for `x`. If the value of `x` is `` `koshka' ``, after the execution of this contraction `x` becomes `` `oshk' ``, and a new variable `s.1` becomes defined and having the value `` `k' ``. If `x` is `` `kot' `` the result is the impossible operation **Z**(failure of matching). Further, this contraction can be decomposed into a product of *elementary* contractions:

$$(\texttt{x}\overset{c}{\rightarrow}\texttt{s.1 x`a'}) = (\texttt{x}\overset{c}{\rightarrow}\texttt{s.1 x})(\texttt{x}\overset{c}{\rightarrow}\texttt{x s.2})(\texttt{s.2}\overset{c}{\rightarrow}\texttt{`a'})$$

The third operation used in Refal graphs is a *restriction* (`#` G), where G is a sum of contractions. It is evaluated either to **Z**, if at least one contraction in G is successful, or to the identity operation **I**(do nothing). An example:

$$(\texttt{`b'}\overset{a}{\leftarrow}\texttt{s.5})\ (\texttt{\# (s.5}\overset{c}{\rightarrow}\texttt{`a')+(s.5}\overset{c}{\rightarrow}\texttt{`b'))} = \mathbf{Z}$$

I developed a set of relations of equivalency for the algebra of pattern-matching operations, and the programming of SCP-3 was based on it, but, unfortunately, this theory is not yet published; the initial stages of the theory can be found in [39].

The most important equation is the *clash* between an assignment and a contraction for the same variable, which is *resolved* in a matching:

$$(e \overset{a}{\leftarrow} v)(v \overset{c}{\rightarrow} p) = e\ :\ p$$

Thus the example above can be seen as the equation:

$$(\texttt{`koshka'}\overset{a}{\leftarrow}\texttt{x})\ (\texttt{x}\overset{c}{\rightarrow}\texttt{s.1 x `a'}) = (\texttt{`oshk'}\overset{a}{\leftarrow}\texttt{x})\ (\texttt{`k'}\overset{a}{\leftarrow}\texttt{s.1})$$

Here is an example of resolution which includes the contraction of a bound variable:

$$(\texttt{`kot's.2}\overset{a}{\leftarrow}\texttt{x})\ (\texttt{x}\overset{c}{\rightarrow}\texttt{x`a'}) = (\texttt{s.2}\overset{c}{\rightarrow}\texttt{`a'})\ (\texttt{`kot'}\overset{a}{\leftarrow}\texttt{x})$$

To give an example of a function definition in the graph form, here is the graph for the iterative program of changing each `'a'` to `'b'`, where nodes $[n]$ correspond to configurations of the Refal machine:

```
[1] ([]<-a- y) [2]
[2] (x-c-> s.1 x) { (s.1-c->'a') ('b'y <-a- y) [2]
                  +(# s.1-c->'a') (s.1 y <-a- y) [2] }
   +(x-c-> []) []<-a- out
```

With Refal graphs, driving is an application of the commutation relations for pattern-matching operations. A walk in the normal form, in which it appears in function definitions and represents one step of the Refal machine, has the structure CRA, where the letters stand for Contraction, Restriction and Assignment, respectively. A walk representing two steps is $C_1R_1A_1C_2R_2A_2$. Resolving the clash A_1C_2 and then adjusting contractions and restrictions according to commutation relations, we return the walk to the normal form: we have made one step of driving.

2.2 Generalization

Generalization is one of the central problems of supercompilation. It breaks down naturally in two parts: (1) a decision to generalize the current configuration C with some of its predecessors C' trying to reduce C to it; we need a 'whistle' to warn us that if we do not try reduction, we may end up with infinite driving; and (2) the generalization proper, t.e. defining a configuration C_{gen} such that both C and C' are its subsets. A good generalization algorithm must find a balance between two extreme cases: a too willing generalization, which makes the resulting program interpretive and leaves it unoptimized; and a generalization postponed for too long so that the supercompilation process never ends.

In [42] I defined an algorithm of whistling for lazy supercompilation of nested function calls, and proved its termination. This algorithm was implemented in SCP-2 and worked very well, but only within its domain of applicability. It gave nothing for generalization of *flat* configurations, i.e. ones without nested function calls. In SCP-2 and SCP-3 we used several empirically found algorithms of generalizing flat expressions, but no termination theorems were proven.

An important step forward was initiated by Morten Sørensen who suggested that the Higman-Kruskal theorem about homeomorphic embedding (HK for short) should be used to define a whistle for supercompilation which is of proven termination. This was done in [32].

The data domain in [32] is the set of functional terms with fixed arity of each functional symbol. This kind of data is sufficient for the languages, such as Lisp and Prolog, which operate on binary trees referred to as lists. But the data structure of Refal does not belong to this category. It allows concatenation of terms, which can be seen as the use of a functional symbol of arbitrary arity (a *varyadic* symbol). Fortunately, the Higman-Kruskal theorem allows variadic symbols. When I learned this, I defined a whistle

for supercompilation in Refal along the line of [32].

I will briefly outline the concept of embedding following the work by Dershowitz [5].

Let a finite set F of functional symbols be given. Consider the set $T(F)$ of all functional terms $f(t_1, \ldots, t_n)$ with functional symbols $f \in F$. Some symbols may be of arity $n = 0$; they are *constants*, and we shall write them without parentheses: f for $f()$. Some of the symbols may be varyadic: different n in different calls.

Definition: The homeomorphic embedding relation $\trianglelefteq$ on a set $T(F)$ of terms is defined recursively as follows:

$$t = g(t_1, t_2, \ldots, t_n) \trianglelefteq f(s_1, s_2, \ldots, s_m) = s$$

if either

$$t \trianglelefteq s_i \quad \text{for some } i = 1, \ldots, m$$

or

$$f = g \quad \text{and} \quad t_j \trianglelefteq s_{i_j} \quad \text{for all } j = 1, \ldots, n$$

where $1 \leq i_1 < i_2 < \ldots < i_n \leq m$. $\Box$

Note that in the second rule f and g must be identical, but their calls may have different number of terms: $n < m$: some of the terms in f may be ignored.

Theorem . (Higman,Kruskal) If F is a finite set of function symbols, then any infinite sequence $t_1, t_2, \ldots$ of terms in the set $T(F)$ of terms over F contains two terms t_j and t_k, where $j < k$, such that $t_j \trianglelefteq t_k$. $\Box$

To use HK, we map the set of all Refal terms T_R onto the domain of functional terms $T(F)$ over some set F of functional symbols.

Definition: The set F is $S \cup \{s_e, s_s, f_{par}, f_{fun}\}$, where S is the set of Refal symbols; it is finite because only those symbol that enter the program can appear in computation. Symbols from S, as well as the two special symbols s_e and s_s, are of arity 0, while the other two symbols are varyadic. The mapping $M_{rt} : T_R \to T(F)$ is recursively defined by:

$M_{rt}[\texttt{s.i}] = s_s$
$M_{rt}[\texttt{e.i}] = s_e$
$M_{rt}[s] = s$
$M_{rt}[(e)] = f_{par}(M_{rt}[e])$
$M_{rt}[\texttt{<}s\ e\texttt{>}] = f_{fun}(M_{rt}[s]\ M_{rt}[e])$
$M_{rt}[t\ e] = M_{rt}[t]\ M_{rt}[e]$
$M_{rt}[\,] =$empty

where s stands for any Refal symbol, t term, and e expression. □

In Refal the most general data structure is that of expressions, not terms. But it is easy to reduce a relation on expressions to a relation on terms:

$$e_1 \trianglelefteq_R e_2 \Leftrightarrow (e_1) \trianglelefteq_R (e_2)$$

We define the homeomorphic embedding relation $\trianglelefteq_R$ on Refal terms as the mapping of the relation $\trianglelefteq$:

$$t_1 \trianglelefteq_R t_2 \text{ if and only if } M_{rt}[t_1] \trianglelefteq M_{rt}[t_2]$$

Rewriting the definition of $\trianglelefteq$ in terms of Refal, we have:

Definition: The homeomorphic embedding relation $\trianglelefteq_R$ on the set T_R holds if either *term is embedded in term*:

$$t \trianglelefteq_R (e_1 t' e_2) \tag{1}$$

where $t \trianglelefteq_R t'$ and e_i for $i = 1, 2$ are some expressions; or *expression is embedded in expression*:

$$(t_1\ t_2 \ldots t_n) \trianglelefteq_R (e_1\ t'_1\ e_2\ t'_2 \ldots e_n\ t'_n\ e_{n+1}) \tag{2}$$

where $t_i \trianglelefteq_R t'_i$ for $i = 1, \ldots, n$, and any of the expressions e_i may be empty. □

This definition translates into the algorithm in Refal given in Table 1.

This algorithm was not yet tested in the computer, but I believe it will work well.

I discovered, with some surprize, that the embedding relation on Refal expresions leads to a whistle different from that derived from the embedding relation on Lisp's lists, even though these two kinds of symbolic objects may look identical. I cannot go into detail here (see [47] for that). Just an example and a few words.

```
* <Emb t1 t2> results in T if t1 ⊴R t2,
* and in F otherwise.
Emb {eX = <Dec eX <Embk 1 eX>> }

* Decide if the second case must be considered
Dec { eX T = T;
      eX F = F;
      eX 2 = <Embk 2 eX> }

Embk {
  sN s1 s1 = T;
  sN t1 s2 = F;
  sN ()(e1) = T;
  1 t1 () = 2;
  2 t1 () = F;
  1 t1 (t2 e3), <Emb t1 t2>:
                { T = T;
                  F = <Embk 1 t1 (e3)> };
  2 (t1 e2)(t.1s e.2s), <Emb t1 t.1s>:
                        {T = <Embk 2 (e2)(e.2s)>;
                         F = <Embk 2 (t1 e2)(e.2s)> };
    }
```

Table 1: The Refal program for the embedding relation $\trianglelefteq_R$.

Consider this relation:

$$(a\ b\ c) \trianglelefteq (a\ p(b\ c)q)$$

If it is understood as a relation between Refal expressions $\trianglelefteq_R$, it does not hold. But if we see it as a relation between lists, it does hold. Moreover, the algorithm for Refal expressions above is linear with the size of expressions, while the corresponding recursive algorithm for lists is exponential (it can be converted, though, into a quadratic iterative algoritm). I must also notice that we can have the Refal whistle working in the Lisp environment by exploiting a one-to-one mapping between these two domains.

A few words about generalization proper. With a given whistle algorithm, various algorithms of the generalization proper may be used. For a whistle algorithm all variables of a given type (s or e) are the same, while for generalization proper this is not so, of course. Still the the embedding

relation which caused the whistle may serve as a starting point for generalization. In particular, if the relation $t_i \trianglelefteq_R t_i'$ in the case *expression imbedded in expression* happens, for some i, to be an equality, we can leave it as a common part in generalization. For example, the embedding:

$$a\ b\ c\ \trianglelefteq_R p\ a\ b\ c\ q\ r$$

leads to the generalization:

$$gen[a\ b\ c,\ p\ a\ b\ c\ q\ r] = x_1 a\ b\ c\ x_2$$

where x_1 and x_2 are some e-variables.

This method, though, works only for Refal's data, not for Lisp's. With lists as the basic data structures, generalization can preserve only that common substructure which is on the left, but not on the right side. Even though a list, such as $(a\,b\,c)$, looks like a string, it is, in fact, a binary tree which in the Refal representation is

$$(a(b(c\,nil)))$$

We can generalize it with a list which extends it on the right side, without loosing the common part:

$$gen[(a(b(c\,nil))), (a(b(c(p\,nil))))] = (a(b(c\,x_1)))$$

but if the extension is on the left, the most specific generalization is a free variable. The common part is lost:

$$gen[(a(b(c\,nil))), (p(a(b(c\,nil))))] = x_1$$

Unfortunately, when a program works by iterations (as opposed to programs where data is passed from the value of one function to the argument of another) it is exactly on the left side that the lists are growing. Because of this, a supercompiler working with lists may not perform partial evaluation in cases where Refal supercompiler easily does it.

2.3 Theorem proving

Using formal logic is not the only way to prove theorems in computers, especially those of primary interest for computer scientists. Metacomputation provides an alternative method of automated theorem proving. We face here two different paradigms.

In the axioms-and-logic paradigm of mathematics we deal with things which are completely undefined, true abstractions. We can know about these things only as much as we can extract from the axioms we have chosen to assume. Mathematical objects – as long as the mathematician is faithful to the proclaimed axioms-and-logic paradigm, which very often is not the case – do not really exist. The meaning of the statement that certain mathematical objects exist is simply that the manipulation of the symbols representing these objects in accordance with the rules of logic and the axioms does not lead to a contradiction.

In contrast, when we are doing computer science we deal with well defined finite cybernetic systems, such as Turing machines or computers, and with computational processes in these systems.

Compare the treatment of the primary theoretical objects of all exact sciences – natural numbers – in the axiomatic and cybernetic paradigms. In axiomatic arithmetics, numbers are abstract entities operations on which meet requirements codified in a certain number of axioms. In particular, the operation of addition $+$ is defined by two axioms:

$$x + 0 = x$$
$$x + y' = (x + y)'$$

where x' is the function 'next number' applied to x. To prove a proposition one must construct, according to well-known rules, a *demonstration*, which is a sequence of propositions.

In the cybernetic paradigm natural numbers are chains of some pieces of matter called symbols, and functions are machines which know how to handle symbols. The Refal program for $+$ is:

```
<+ x,'0'> = x
<+ x,y'1'> = <+ x,y>'1'
```

Another function we want to compute is the predicate of equality:

```
<= '0','0'> = T
<= '0',y'1'> = F
<= x'1','0'> = F
<= x'1',y'1'> = <= x,y>
```

The strong side of the axioms-and-logic method is its wide applicability. A theory may be developed about objects (such as those of geometry or set theory) which are not easy to represent by symbolic expressions. Also, the same theorem can be used with different interpretations. Group theory

is usually adduced as an example. However, in computer science it is exactly the world of symbolic expressions and processes that we are primarily interested in. For this world the cybernetic paradigm is pretty natural.

The obvious advantage of the cybernetic paradigm is the completely mechanized way to perform computations. To prove that $2+2=4$, we only have to compute the truth-value of the proposition:

```
<= <+ '011','011'>,'01111'>
```

When we give this job to the Refal machine, the results is a finite computation process. It ends with `T`, which proves the statement.

This has been a proposition without quantification. Can general propositions be proven by computation?

Not directly. But they can be proven by *metacomputation*. For any tautology of the propositional calculus metacomputation is straightforward and guarnateed to give the desired proof. Define logical connectives `NOT`, `AND`, etc.:

```
<NOT T> = F
<NOT F> = T
```

and the others in the same manner. To prove de Morgan's law $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$, form the configuration:

```
[1] <EQU <NOT <AND p,q>> , <OR <NOT p>,<NOT q>>>
```

It includes two free variables `p` and `q`, but all the functions involved are not recursive, so the simple driving gives the finite result:

```
[1]; (p -c-> F)(q -c-> F) T
+ (p -c-> F)(q -c-> T) T
+ (p -c-> T)(q -c-> F) T
+ (p -c-> T)(q -c-> T) T
```

Now we only have to recognize that all exit configurations in this graoh are `T`, hence the configuration `[1]` can be replaced by `T`. This amounts to the desired proof.

Let us take a simple example from arithmetics where the proof requires the use of mathematical induction. Consider the following statement:

$$\forall x\,(0+x=x) \tag{3}$$

Let us see how this theorem is proven by a supercompiler. The translation of the statement into the cybernetic paradigm is as follows. The initial configuration:

```
[1]:  <= <+ '0',x>, x>
```

evaluated by the Refal machine with any value substituted for `x` results in `T`; so we expect from the supercompiler that it will equivalently transform `[1]` into just `T`.

The supercompiler that makes this job uses the outside-in (lazy) driving. It attempts to drive the call of `=`, but the nested call of `+` is a hindrance, thus it switches to driving the latter. Two contractions are produced in accordance with the definition of `+`: `x`$\stackrel{c}{\rightarrow}$`'0'` and `x`$\stackrel{c}{\rightarrow}$`x'1'`. With the first contraction the computation is straightforward and leads to `T`. With the second contraction the machine makes a step in the computation of `<+ '0',x'1'>`, which produces the configuration:

```
[2]:  <= <+ '0',x>'1', x'1'>
```

The process returns to the outermost call and makes, in a unique way, one step according to the definition of `=`. The result is the same as `[1]`. Thus we have this transition graph:

```
[1] x -c-> '0'; T
  +  x -c-> x'1'; [1]
```

The supercompiler easily recognizes such a graph as transformable to just `T`, since the only exit configuration is `T`; this is the form which mathematical induction takes in metacomputation. The theorem is proven. (We set aside the problem of termination. In our case it is secured by the fact that all functions involved are total).

The associativity of addition is also easily provable in this way. However, to prove the commutativity of addition requires more sophisticated techniques, which I shall discuss in Sec. 3.1.

2.4 Metasystem hierarchies and jumps

Consider metasystem hierarchies of computing machines. After we have chosen a universal all-level programming language, such as Refal, a hierarchy of computing machines becomes a hierarchy of programs. We want to write functions which are defined on definitions of other functions.

In every programming language we distinguish the *objects* which are manipulated, from certain special details, variables and function calls, which

e	$\mu\{e\}$
`[]`	`[]`
s	s
`s.`i	`('s'`i`)`
`e.`i	`('e'`i`)`
`(`e`)`	`('*'`$\mu\{e\}$`)`
`<`e`>`	`('!'`$\mu\{e\}$`)`
$e_1 e_2$	$\mu\{e_1\}\mu\{e_2\}$

Table 2: The metacode

represent sets of objects and computation processes and cannot be directly treated as objects. Let the set of objects be S_{ob} and the set of variables and function calls S_{vf}. To write a program which manipulates programs, we must map the set of all elements of programs, i.e. $S_{ob} \cup S_{vf}$, on the set of objects S_{ob}. We call this mapping a *metacode*, and denote the metacode transformation of e as $\mu\{e\}$:

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

Obviously, metacoding must have a unique inverse transformation, *demetacoding*, so it must be injective:

$$\forall (e_1, e_2)\, e_1 \neq e_2 \rightarrow \mu\{e_1\} \neq \mu\{e_2\}$$

For convenience of reading metacoded expressions we require that $\mu\{e_1 e_2\} = \mu\{e_1\}\mu\{e_2\}$. Also, it is 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.

One convenient metacode for Refal, which is used in the latest implementation of this language is defined by Table 2, where s is any symbol, and i the index of a variable.

Consecutive metacoding creates a hierarchy of *MST domains*:

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

where $S^0 = S_{ob} \cup S_{vf}$, and $S^k = \mu\{S^{k-1}\}$, for $k \geq 1$.

Compare two function calls: `<F2 <F1 x>>` and `<F2` $\mu\{$`<F1 x>`$\}$`>`. The first call is a functional composition: `<F1 x>` is computed and its value is taken as argument in the computation of `F2`. In the second call there is no evaluation of `<F1 x>`, but the metacode of this expression is turned over for the computation of `F2`. If the metacode of Table 2 is used, we have: `<F2 ('!'F1('ex'))>`. This is an MST hierarchy of two levels: function `F2` is supposed to manipulate `F1` (its representation, to be precise). If this manipulation is *semantic* in nature, i.e. based on the definition of `F1`, as it is in all interesting cases, then the machine `F2` must have access to the definition of `F1`. In order not to encumber our notation, we shall always assume that whenever a metamachine F_2 manipulates a machine F_1, it incorporates the definition of F_1, so we need not indicate this in each case.

We use *MST schemes* for a clear and metacode-invarient representation of metasystem hierarchies. 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:

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

This rule can be applied any number of times.
Refal expressions on the lower level are written the same way as if they were on the upper level; metacoding is implicit and is indicated by putting them one line down. To convert an MST scheme into an equivalent Refal expression, we must metacode each level as many times as long is its distance from the top.

MST schemes allow us to represent very clearly certain operations on variables which are necessary for correct construction and use of metasystem hierarchies. We shall show this using as an example the well-known procedure of converting an interpreter for some language into a compiler by partial evaluation (see [8], [40], [17]). Let `L` be an interpreter for some language L written in Refal and used in the format `<L e.prog,e.data>`.

Let `PE` be a partial evaluator for Refal written in Refal and having Refal as the target language, i.e. producing a Refal program at the output. We apply `PE` to the call of `L` where some program P is substituted for `e.prog`, while `e.data` remains free. This call is `<L` P`,e.data>`. We metacode it and submit to the partial evaluator:

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

In this MST scheme the call of L, which is submitted for partial evaluation, is a function of data only, since the value of e.prog is fixed at a specific expression P. After PE performs all operations which can be performed because the program P is known, it outputs a residual program which is nothing but the translation of the program P into Refal. Function PE has worked as a compiler.

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

```
<PE ..................... >
    <L e.prog, 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. No partial evaluation takes place, because the value of e.prog remains unknown to PE. Even though e.prog is an argument of L, its value 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 e.prog to the top level, and leaving the bullet • in the place where this variable originated on the bottom level:

```
<PE .. e.prog ........... >
    <L     •     ,e.data>
```

We shall call the variables like e.prog *elevated*. For such a variable, the definition level, at which its name is placed, is different from the usage level, where the bullet is found, and the difference h between the two is the variable's *elevation*. The value assigned to a variable on the definition level enters the configuration after being metacoded h times.

Even though e.prog 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 e.prog. Hence L always receives a fixed program. The result of PE will be a transformed (partially evaluated) function L, which depends only on the variable e.data and is a translation from L into Refal.

The translation, however, is made directly by PE, which can work with any definition of L (hidden in the function name L, as we agreed above). We can specialize PE by partially evaluating itself by itself according to the MST scheme:

```
<PE ........................ >
    <PE .. e.prog ......... >
        <L     •    ,e.data>
```

This scheme is the scheme of generation of a compiler from L. The

program which it produces is a compiler: it has a program at input and a function of data at output.

The *rule of two levels* helps read MST schemes: The variables on the top level are *free*. Those on the next level are *bound*: they run over their domains as, e.g., integration variables, or the variables in a function definition. Any top part can be chopped off from the rest of the scheme; hence every level can be interpreted as if it were the top.

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. The interpreter `L` operates on a known program andunknown data. On some stretches of computation `L` will work on the program, but without consulting the data. An obvious example is the 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 than a direct run of the function `L`.

In [45] we describe a supercompiler which makes automatic jumps from one level to another in order to avoid doing on the level n in the interpretation mode what can be done on the level $n-1$ by direct computation. Before driving a configuration, the supercompiler passes control one level down by demetacoding the configuration and starting the execution of it. If it is possible to bring the computation to the end, the result is metacoded and control returns to the upper level. If at a certain stage of execution its continuation becomes impossible because of unknown values of variables, the configuration of the latest stage preceding the current is metacoded and controll passes to the top level for driving.

To make this system work, it was necessary to modify the implementation of Refal by adding a feature which was called a *freezer*, see [43]. In tests we could see that the use of metasystem jumping can lead to a very considerable speedups, sometimes by a factor of more than twenty.

2.5 Refal *vs.* Lisp

As mentioned above, the major difference between Refal and Lisp, as well as other languages working with lists, is the data domain. Refal expressions are strings of terms which can be processed both left to right, and right to left. They are trees with an arbitrary and unfixed arity. Lisp's list is a special kind of a Refal expression. There are several reasons why I stubbornly use Refal (beyond the main reason, which I am trying to conseal: I invented it).

1. I hope that sooner or later the methods of metacomputation will be used on the industrial scale for automatic development of big and fast programs. The efficinecy of algorithms is, as we very well know, tightly bound to data structure. I cannot imagine that practical programmers will agree to abandon such an important data structure as string. Limiting ourselves to lists, we throw overboard a huge pile of efficient algorithms.

2. As we saw above, Refal expressions allow such generalizations which are inexpressible in lists. This makes supercompilation easier and more efficient. Our data structures are, essentially, models of reality. More sophisticated data structures allow us to express more subtle features of the medium. A language based on graphs would have the same advantage over Refal as Refal has over Lisp.

3. When we create a metamachine for examining and controlling the operation of the object machine, we often want to trace its steps in both forward, and backward directions. Histories of computation are naturally represented by strings of states. In Sec.3.1 I show how supercompilation can be enhanced by switching from configurations of the computing machine to histories of computation by it. Moreover, backward movement is part of the concept of supercompilation. It can be avoided, but not without paying some price – in conceptual simplicity, if not in anything else. Languages we use not only help express somehting we are doing; they suggest what we might do further. It is not an accident that the concept of supercompilation first appeared in the context of such a language as Refal.

4 Functional languages using lists are good for programming in recursive style, but poor when the algorithms are iterative. Meanwhile, we often face the situation where a recursive algorithm is clear and elegant, but inefficient, so that we have to transform it into an iterative form. In Lisp even a simple traversal without inverting the list is impossible when we do it iteratively. Refal is equally at ease with both recursive and iterative programming. □

The use of lists, though, is not without its own advantages. As a data structure for analysis and manipulation, lists are simpler than Refal expressions, though in my view, the difference is not significant. Another advantage is the simple fact that people in computer science research have accustomed to this domain, and the languages based on it are widely used. One balances these two sets of advantages according to one's priorities.

3 Still to come

3.1 Walk grammars

Now I will show how to make one more – and, maybe, most promising – metasystem transition in program transformation, [39, 44].

As discussed in Sec.2.1, one step of the Refal machine is represented in the pattern-matching graph as a normal walk $C_1R_1A_1$. For driving we combine two steps by concatenating two normal walks: $C_1R_1A_1C_2R_2A_2$, and normalize them into one walk again. We can postpone driving and combine any number of elementary walks into unnormalized walks of arbitrary length. Such walks will represent possible (but not necessarily feasible) histories of computation without performing the computation itself.

Take the graph

```
[1] ([]⟸a y) [2]
[2] (x⟹c s.1 x) { (s.1⟹c'a') ('b'y ⟸a y) [2]
                  +(# s.1⟹c'a') (s.1 y ⟸a y) [2] }
   +(x⟹c []) []⟸a out
```

from Sec.2.1. Denote the one-step walks in the graph as follows:

$w_1 =$ `([]⟸a y)`
$w_2 =$ `(x⟹c s.1 x) (s.1⟹c'a') ('b'y ⟸a y)`
$w_3 =$ `(x⟹c s.1 x) (# s.1⟹c'a') (s.1 y ⟸a y)`
$w_4 =$ `(x⟹c []) []⟸a out`

Now the set of all terminated walks (histories of completed computation) is described by the regular grammar:

`[1]` $\Rightarrow w_1$ `[2]`
`[2]` $\Rightarrow w_2$ `[2]`
`[2]` $\Rightarrow w_3$ `[2]`
`[2]` $\Rightarrow w_4$

or by the regular expression $w_1(w_2 + w_3)^*w_4$.

It is easy to see that for an arbitrary Refal program the set of all walks is defined by a context-free grammar, while if we restrict ourselves to *flat* Refal (no nested calls) the walk grammar becomes regular, hence the whole set is represented by a regular expression, which is a great advantage of the flat version. We shall work with a generalization of regular expressions, where the number of iterations is denoted by a variable, so that such walk-sets as $w^n w^n$ are permitted.

Walk-sets are an alternative form of a program. We can define an interpreter Int which executes such programs, and transform Int calls by the supercompiler: an MST to the three-level hierarchy: $Scp - Int - Program$. But what do we achieve by this MST? Well, if Int does just driving, i.e. processes the walks in the strict order from left to right, we have achieved nothing. But we can create a clever interpreter which uses equivalency relations on the set of walks. It could *unwind* iterative loops both from the left, and from the right, i.e. use the relation $W^{n+1} = W\,W^n$ or $W^{n+1} = W^n W$ in order to achieve reduction in supercompilation. It could also reorder operations in walks using commutation relations and do other transformations. In [44] I show that with this technique we can make transformations which could not be done by direct supercompilation, such as function inversion and the merging of consecutive iterative loops.

The $Scp - Int$ technique can be very effectively used for *graph cleaning*. Suppose we finished supercompilation and have the resulting flat graph G. It does no mean that all exit nodes (expressions for output) are actually feasible. For each node in G we can write a regular expression for the set of all walks which lead to this node. Using the Scp-Int method we may discover that some walk-sets are reduced to $\mathbf{Z}$; then these nodes can be eliminated.

As an example, let us return to the proof of commutativity of addition, which we have found more difficult to prove than the other theorems in Sec.2.3. The proof requires a considerable analysis of the graph resulting from straightforward supercompilation. I will only sketch the proof.

The configuration to compute is

```
[1] <= <+ x,y>, <+ y,x>>
```

In the process of supercompilation, `[1]` is generalized, and reduced to the generalization:

`[1]` `(x` $\overset{a}{\Leftarrow}$ $\bar{\texttt{x}}$`)(y` $\overset{a}{\Leftarrow}$ $\bar{\texttt{y}}$`)` `[2] <= <+` $\bar{\texttt{x}}$`,y>, <+` $\bar{\texttt{y}}$`,x>>`

The graph for `[2]` has 15 exits. Four of them are `T`, eleven are `F`. To prove the theorem, the walks which lead to each of the eleven `F` ends must

be proven unfeasible. As an example, I will do it for one of the walks, which is neither the least nor the most difficult:

$$w = \texttt{(x}\overset{a}{\leftarrow}\bar{\texttt{x}}\texttt{)(y}\overset{a}{\leftarrow}\bar{\texttt{y}}\texttt{)(y}\overset{c}{\rightarrow}\texttt{y`1')}^n\texttt{(x}\overset{c}{\rightarrow}\texttt{x`1')}^n\texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')(x}\overset{c}{\rightarrow}\texttt{`0')}$$
$$\texttt{(}\bar{\texttt{y}}\overset{c}{\rightarrow}\bar{\texttt{y}}\texttt{`1')(}\bar{\texttt{y}}\overset{c}{\rightarrow}\bar{\texttt{y}}\texttt{`1')}^m\texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')}^m\texttt{(}\bar{\texttt{y}}\overset{c}{\rightarrow}\texttt{`0')(y}\overset{c}{\rightarrow}\texttt{y`1')}$$

Using commutation relations we reduce it to the form:

$$w = \texttt{(x}\overset{a}{\leftarrow}\bar{\texttt{x}}\texttt{)}\ \texttt{(x}\overset{c}{\rightarrow}\texttt{x`1')}^n\ \texttt{(x}\overset{c}{\rightarrow}\texttt{`0')}$$
$$\texttt{(y}\overset{a}{\leftarrow}\bar{\texttt{y}}\texttt{)}\ \texttt{(}\bar{\texttt{y}}\overset{c}{\rightarrow}\bar{\texttt{y}}\texttt{`1')}^p\ \texttt{(}\bar{\texttt{y}}\overset{c}{\rightarrow}\texttt{`0')}\ \texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')}^q$$

where we have introduced new variables p, q, which show the total number of iterations in the walk. Their relation to the loop variables m, n, namely $p = m + 1$ and $q = n + m + 2$, will be treated as a restriction on p, q.

The unfeasibility of w results from the second line, i.e. y-part of it; so we ignore the first line (x-part). The call of the interpreter takes the form:

$$\texttt{<Int (y}\overset{a}{\leftarrow}\bar{\texttt{y}}\texttt{)}\ \texttt{(}\bar{\texttt{y}}\overset{c}{\rightarrow}\bar{\texttt{y}}\texttt{`1')}^p\ \texttt{(}\bar{\texttt{y}}\overset{c}{\rightarrow}\texttt{`0')}\ \texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')}^q\texttt{>}$$

This is a function of p and q. Unwinding the p-loop means breaking it into a resursive call and the base:

$$W^p = (p \overset{c}{\rightarrow} p+1)\ WW^p\ +\ (p \overset{c}{\rightarrow} 0)\quad \texttt{[]}$$

and analogously for the q-loop.

Thus, driving produces branches to four cases:

```
[1] <Int w>  (p→ p+1)(q→ q+1) [2]
           + (p→ p+1)(q→'0') [3]
           + (p→'0')(q→ q+1) [4]
           + (p→'0')(q→'0') [5]
```

Let us consider the walk transformation in `[2]`. This is the case when recursion takes place in both p-loop, and q-loop:

$$\texttt{(y}\overset{a}{\leftarrow}\bar{\texttt{y}}\texttt{)(}\bar{\texttt{y}}\overset{c}{\rightarrow}\bar{\texttt{y}}\texttt{`1')(}\bar{\texttt{y}}\overset{c}{\rightarrow}\bar{\texttt{y}}\texttt{`1')}^p\ \bar{\texttt{y}}\overset{c}{\rightarrow}\texttt{`0')}\ \texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')(y}\overset{c}{\rightarrow}\texttt{y`1')}^q$$

Resolving the clash on $\bar{\texttt{y}}$, we have:

$$\texttt{(y}\overset{a}{\leftarrow}\bar{\texttt{y}}\texttt{)}\ \texttt{(}\bar{\texttt{y}}\overset{c}{\rightarrow}\bar{\texttt{y}}\texttt{`1')}\ \texttt{=}\ \texttt{(y:}\bar{\texttt{y}}\texttt{`1')}\ \texttt{=}\ \texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')}\ \texttt{(y}\overset{a}{\leftarrow}\bar{\texttt{y}}\texttt{)}\ \texttt{(y`1'}\overset{a}{\leftarrow}\texttt{y)}$$

Now the assignment $\texttt{(y`1'}\overset{a}{\leftarrow}\texttt{y)}$ travels to the right and clashes with contraction $\texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')}$ to produce nothing. The contraction $\texttt{(y}\overset{c}{\rightarrow}\texttt{y`1')}$ is taken from the argument of `Int` and becomes a contraction on the input variable $y$ (I cannot go here into formal details). The result is that `[2]` becomes identical to `[1]`: a reduction (folding).

It is easy to check, that configurations `[3]` and `[4]` are unfeasible. Configuration `[5]` is an exit from the loop where p and q are decreased by one in every cycle. If we denote the number of cycles as N, the restriction on p, q becomes:

$$p + N = m + 1, \quad q + N = n + m + 2$$

When $p = q = 0$, we have the restriction $m + 1 = n + m + 2$, which cannot be satisfied because $n \geq 0$. The graph for `[1]` becomes a loop without exits:

`[1] <Int w> (p`$\stackrel{c}{\rightarrow}$` p+1) (q`$\stackrel{c}{\rightarrow}$` q+1) [1]`

This completes the proof of unfeasibility of the chosen walk. The other ten walks are handled analogously.

3.2 Alternating quantifiers

The logical formula for commutativity of addition is universally quantified over x and y. Now we want to find out how the computational paradigm tackles the cases of existential quantification and, especially, those where $\forall$ and $\exists$ alternate. As is well known, a sequence of identical quantifiers can be reduced to one by operating on tuples of variables, but there is no similar reduction when quantifiers alternate. We shall see that each putting of $\forall$ in front of $\exists$, or vice versa, requires, computationally, a metasystem transition.

Let `All` be a function which uses a supercompiler to prove universally quantified statements, as in the above examples. If the supercompiler comes with a graph where no branch ends with `F`, it outputs `T`, otherwise it outputs `Z`. Thus

$$\texttt{<All } \mu\texttt{<P x>>} = \begin{cases} \texttt{T};\ \forall x\, P(x) \text{ is proven} \\ \texttt{Z};\ \text{no information} \end{cases}$$

To introduce existential quantification we define the function `Exs` which constructs by driving the potentially infinite tree of configurations using the breadth-first principle. If it finds that some branch ends with `T`, it outputd `T`. Otherwise it works infinitely – or, realistically, till it is stopped:

$$\texttt{<Exs } \mu\texttt{<P x>>} = \begin{cases} \texttt{T};\ \exists x\, P(x) \text{ is proven} \\ \text{is stopped; no information} \end{cases}$$

As always in Refal, a variable `x` may be an n-tuple `(x1)`...`(xn)`. If all of them take part in the driving implied in `<All` μ`<P x>>` or `<Exs` μ`<P x>>`, they are all appropreately quantified, e.g. computing

```
<All ..... >
     <P x, y, z>
```

is proving $\forall x \forall y \forall z P(x,y,z)$. We can fix the value of one variable, say `x`, by raising it to the top level (see Sec.2.4):

```
<All ...x ..... >
     <P •, y, z>
```

This computation requires some value of x be given, and for this value it tries to prove $\forall y \forall z P(x,y,z)$.

We can consider this function as a predicate depending on x and quantify it existentially by submitting it to function `Exs`:

```
<Exs ..................... >
   <All ...x ..... >
        <P •, y, z>
```

This is the metacomputation formula (MST scheme) for $\exists x \forall y \forall z P(x,y,z)$. In a similar manner we establish that the logical formula $\forall x \exists y \forall z P(x,y,z)$ is represented by:

```
<All ..................... >
  <Exs ..... x ........... >
     <All .. |  y ..... >
          <P •, •, z>
```

Following these lines it is easy to construct an MST scheme for every logical formula, after it has been reduced to the prenex form.

As an example, consider the theorem: there exists no maximal natural number: $\neg \exists x \forall y Less(y,x)$. Here the function `Less` is defined as follows:

```
<Less x'1',y'1'> = <Less x,y>
<Less x'1','0'> = F
<Less '0',y'1'> = T
<Less '0','0'> = F
```

To prove the theorem, we first reduce the proposition to the prenex normal form: $\forall x \exists y Less(x,y)$, (we have used the equivalence $\neg L(y,x) = L(x,y)$), then we form the corresponding MST scheme:

```
<All ............... >
     <Exs ..... x .. >
          <Less •, y>
```

Supercompilation of this simple configuration can be done manually, and I did this. I wrote a specialized version of `Exs` which does driving

in the expectation that the only function called is `Less`. Then I did the supercompilation implied in `All`, and the result was `T`, which proves the theorem.

3.3 Neighborhood analysis

Consider a function which looks for the first `'a'` in the string and returns `T` if it is found; otherwise it returns `F`:

```
<fa 'a'x> = T
<fa s.1 x> = <fa x>
<fa []> = []
```

Consider the computation of `<fa 'kasha'>`:

```
1. <fa 'kasha'>
2.  <fa 'asha'>
3.       T
```

One may notice that there is a part of the argument, namely `'sha'`, which did not take part in computation. It could be replaced by any expression, and the computation, as well as its final result, would not change a bit. One might guess that this kind of information about computational processes may be of interest for different purposes, such as debugging and testing programs. A variation of driving, *driving with neighborhood*, [39], provides a general method for representing and extracting such information.

We define a *neighborhood* as the structure $(a)p$, where p is a pattern and a is a list of assignments for all variables in p, such that the result of substitution a/p is a ground expression (i.e. one without variables) referred to as the *center* of the neighborhood. Driving with neighborhood is a combination of computation and driving. In computation the argument is a ground expression, and it defines the path of computation: which sentence is used at each step. In driving the argument is an arbitrary expression, and we analyze all possible for it computation paths. Driving with a neighborhood we drive its pattern, but consider only one path, namely the one taken by the neighborhood's center. At each step the center is the same as if the initial function call were directly evaluated. The free variables in the pattern (which may be loosely called neighborhood) represent the part of information which was not, up to the current stage, used in computation.

Consider this driving:

1.	(‘kasha’$\overset{a}{\leftarrow}$y) y	<fa y>	y$\overset{c}{\rightarrow}$‘k’y
2.	(‘asha’$\overset{a}{\leftarrow}$y) ‘k’y	<fa y>	y$\overset{c}{\rightarrow}$‘a’y
3.	(‘sha’$\overset{a}{\leftarrow}$y) ‘ka’y	T	

In the initial neighborhood (column 2) the pattern has the maximal extension: anything, a free variable ‘y’; the assignment makes the center ‘kasha’. In column 3 is the call to compute the pattern. By driving under the definition of fa, we find that the contraction in column 4 is necessary in order to take the path the center will take (the second sentence of the definition). Modifying the neighborhood by this contraction, we have the next stage neighborhood in line 2. Proceeding further in this manner, we complete the computation of the initial call – it is T – and get the representation of the argument as a neighborhood with the pattern ‘ka’y, which tells us that together with our argument, any argument which matches ‘ka’y will lead to the same result of computation.

Sergei Abramov found a way to use neighborhood analysis for program testing, [1, 2]. This may seem strange, because the neighborhoods give us information about unused parts of arguments, *data*, not about the program. But Abramov makes a metasystem transition: driving with neighborhood is applied not to program P working on data D, but to program Int which is em an interpreter of the language in which P is written and works on the pair (P, D). Now the program becomes data. Fixing some input D – a test for P – we can, by neighborhood analysis, determine what parts of the program were used, and hence tested,in this run, and which parts were not.

On this basis Abramov built an elegant theory of program testing, which usues the principle: choose each next test so as to check those features of the program which have not yet been tested. Abramov gives a precise mathematical definition to this intuitive principle and provides the necessary theorems and algorithms.

4 Conclusion

Supercompilation and partial evaluation belong to the same kind of program transformation, which we refer to as metacomputation. Supercompilation includes, but goes far beyond, partial evaluation. It may cause a deep transformation of a program, especially when combined with various metasystem transitions. Potential applications of driving, supercompilation, and other forms of metacomputation are numerous and include function inversion, program testing and theorem proving. Especially intriguing is the possibility

of repeated metasystem transitions from processing a function to processing the sets of possible computation histories – walks in the graph of this function as shown in Sec.3.1.

The boundaries of what is possible to do with supercompilation are not yet known. I would say that while supercompilation is, certainly, not sufficient for successful program transformation, it is, in a sense, necessary because this concept is a computer implementation of the general principle of human knowledge, which is a search for such generalized states in terms of which we can construct a self-sufficient model of a system. One consequence of this nature of supercompilation is its universality. No special conditions or restrictions are set beyond the fact that we deal with a computing system. In my view, it is natural to find what can be done by supercompilation before trying more complicated approaches. This is what I mean by saying that supercompilation is necessary.

Even though the principle of metacomputation is simple, its translation into working machines may be far from being simple. This is not unusual. The basic principles of flying are also simple, but an airplane consists of many thousands of details, and it has taken many years of work by many people, in order to develop air technology to the stage when the wonderful machines of today become possible. Metacomputation and, in particular, supercompilation are now at the technological stage of the brothers Wright. Let us hope that a steady process of technological improvement of supercompilers can be started, and that it will produce software tools to be widely used in computer science and industry.

Acknowledgment. To avoid repetition of what I have said in the History section, let me simply express my deep gratitude to all those who worked with me or appreciated my work.

References

[1] S.M.Abramov. Metacomputation and program testing, in: *1st International Workshop on Automated and Algorithmic Debugging*, Linkoping, Sweden, pp.121-135, 1991.

[2] S.M.Abramov. *Metavychisleniya i ikh Prilozhenija (Metacomputation and its Computations, in Russian)* Nauka, Moscow, 1995.

[3] *Bazisnyi Refal i yego realizatsiya na vychislitel'nykh mashinakh, (Basic Refal and its implementation on computers, in Russian)*, GOSSTROY SSSR, TsnIPIASS, Moscow, 1977.

[4] S.V.Chmutov, E.A.Gaydar, I.M.Ignatovich, V.F.Kozadoy, A.P.Nemytykh, V.A.Pinchuk. Implementation of the symbol analytic transformations language FLAC, DISCO'90, LNCS vol. 429, p.276, 1990.

[5] Dershowitz,N. Termination in rewriting, *Journal of Symbolic Computation*, 3, pp.69-116, 1987.

[6] Ershov, A.P. On the essence of compilation, *Programmirovanie* (5):21-39, 1977 (in Russian). See translation in: E.J.Neuhold, ed., *Formal description of Programming Concepts* pp 391-420, North-Holland, 1978.

[7] Opening Key-note Speech, in: D.Bjorner,A.P.Ershov and N.D.Jones, ed. *Partial Evaluation and Mixed Computation*, North-Holland, pp.225-282, 1988.

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

[9] Futamura Y., Nogi K. Generalized Partial Evaluation, in: Bjorner D., Ershov A.P., Jones N.D. (eds), *Partial Evaluation and Mixed Computation, Proceedings of the IFIP TC2 Workshop*, pp.133-151, North-Holland Publishing Co., 1988.

[10] Futamura, Y.,Nogi, K., Takano, A. Essence of generalized partial evaluation, *Theoretical Computer Science*, 90, pp. 61-79, 1991.

[11] 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.

[12] R.Glück and J.Jørgensen. Generating transformers for deforestation and supercompilation, in: B. LeCharlier ed. *Static Analysis, Proceedings*, Namur, Belgium, 1994, LNCS, vol.864, pp.432-448, Springer, 1994.

[13] R.Glück and A.V.Klimov. Occam's razor in metacomputation: the notion of a perfect process tree, in: P.Cousot, M.Falaschi, G.File, and Rauzy, ed. *Static Analysis*, LNCS vol724, pp.112-123, Springer 1993.

[14] R.Glück and A.V.Klimov. Metacomputation as a tool for foramal linguistic modelling, in: R.Trapple, ed. *Cybernetic and Systems '94* vol.2 pp.1563-1570, SIngapore, 1994

[15] R.Glück and A.V.Klimov. Metasystem transition schemes in computer science and mathematics, *World's Future: the Journal of General Evolution*, vol.45, pp.213-243, 1995.

[16] R.F.Gurin and S.A.Romanenko *The Programming Language Refal Plus* (in Russian), Intertekh, Moscow, 1991.

[17] Jones N.D., 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, pp.124-140, Springer, 1985.

[18] Jones, N.D. Automatic program specialization: a re-examination from basic principles, in: D.Bjorner,A.P.Ershov and N.D.Jones, ed. *Partial Evaluation and Mixed Computation*, North-Holland, pp.225-282, 1988.

[19] 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.

[20] N.D.Jones, The essence of program transformation by partial evaluation and driving, in: N.D.Jones, M.Hagiya, and M.Sato ed. *Logic, Language and Computation*, LNCS vol.792, pp.206-224, Springer, 1994.

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

[22] Klimov A.V. and Romanenko S.A. *A Meta-evaluator for the language Refal, Basic Concepts and Examples* (in Russian), Preprint 71 Keldysh Institute for Applied Mathematics, Moscow, USSR, 1987.

[23] José Meseguer. General logics, in: H.-D. Ebbinghaus et al ed. *Logic Colloquium'87*, pp.275-329, North-Holland, 1989.

[24] José Meseguer and Manuel Clavel. Axiomatizing reflective logics and languages, submitted for publication.

[25] Olunin V.Yu., Turchin V.F., FlorentsevS.N., A Refal interpreter, in: *Trudy 1-oi Vses. Konf. po Programmirovaniyu*, Kiev, 1968 (in Russian)

[26] P.Sestoft. The structure of a self-applicable partial evaluator, in: H.Ganzinger and N.D.Jones, ed. *Programs as Data Objects (Copenhagen, 1985)*, LNCS, vol.217, pp.236-256, Springer, 1986.

[27] The generation of inverse functions in Refal. in: D.Bjorner,A.P.Ershov and N.D.Jones, ed. *Partial Evaluation and Mixed Computation*, North-Holland, pp.427-444, 1988.

[28] Inversion and metacomputation, in: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Yale University), pp.12-22, ACM Press, 1991.

[29] Romanenko, S.A. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure, in: D.Bjorner,A.P.Ershov and N.D.Jones, ed. *Partial Evaluation and Mixed Computation*, North-Holland, pp.445-464., 1988.

[30] Romanenko S.A. Arity raiser and its use in program specialization, in: Jones N.D. ed., ESOP'90, LNCS, vol.432, pp.341-360, 1990.

[31] M.H.Sørensen. *Turchin's Supercompiler Revisited*, Master's thesis, Dept. of Computer Science, University of Copenhagen, 1994.

[32] M.H.Sørensen, R.Glück. An algorithm of generalization in positive supercompilation, in: J.W.Lloyd ed., *International Logic Programming Symposium*, MIT Press, 1995, to appear.

[33] M.H.Sørensen, R.Glück and N.D.Jones. Towards unifying deforestation, supercompilation, partial evaluation and generalized partial evaluation, in: D.Sannella ed., *Programming Languages and Systems*, LNCS, vol.788, pp.485-500, Springer, 1994.

[34] M.H.Sørensen, R.Glück and N.D.Jones. A positive supercompiler. Submitted to *Journal of Functional Programming*, 1995.

[35] Metajazyk dlja formal'nogo opisanija algoritmicjeskikh jazykov (A metalanguage for formal description of algorithnic languages, in Russian), in: *Cifrovaja Tekhnika i Programmirovanie*, pp.116-124, Moscow 1966.

[36] Turchin V.F. Programmirovanie na yazyke Refal (Programming in Refal, in Russian), Preprints Nos. 41, 43, 44, 48, 49 of the Institute for Applied Mathematics, AN SSSR, 1971.

[37] Turchin, V.F., Equivalent transformations of recursive functions defined in Refal. In: Teoriya Yazykov I Metody Postroeniya Sistem Programmirovaniya (Proceedings of the Symposium), Kiev-Alushta (USSR), pp.31-42, 1972 (in Russian).

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

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

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

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

[42] Turchin, V.F. The algorithm of generalization in the supercompiler. In: Bjorner D., Ershov A.P., Jones N.D. Eds, Partial Evaluation and Mixed Computation, Proceedings of the IFIP TC2 Workshop, pp. 531-549, North-Holland Publishing Co., 1988.

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

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

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

[46] Turchin V., Nemytykh, A. A Self-applicable Supercompiler CCNY Technical Report CSc TR-95-010, 1995.

[47] Turchin, V.F. On Generalization of Lists and Strings in Supercompilation, CCNY Technical Report, 1996.