ISBN 5-94052-066-0 PROGRAM SYSTEMS: THEORY AND APPLICATIONS. PERESLAVL', 2004

UDC 681.32

A. P. Nemytykh

The Supercompiler Scp4: General Structure

Abstract. The author constructed a transformer Scp4 of functional programs. The transformer uses the technology known as Turchin's supercompilation. Scp4 was implemented in a functional language Refal-5. The input language for Scp4 is also Refal-5. In the paper we consider the general structure of the supercompiler and give a number of examples of transformations.

1. Introduction

Supercompilation is a program transformation technique introduced in the 1970s by Valentin F. Turchin [21, 22, 24–26]. He suggested a task of creating tools to observe operational semantics of a program, when a function F that is to be computed by the program is fixed. As a result of such observations a new algorithmic definition of an extension of the function F must be constructed.

The Turchin's ideas were studied by a number of authors for a long time and have to some extent been brought to the algorithmic stage. We constructed an experimental supercompiler for a functional language Refal-5 [23]. The Scp4 project was discussed with V. F. Turchin. Moreover, he initiated and supported our work.

Scp4 has been implemented once again using Refal-5. Sources of the supercompiler, executable modules and sources of Refal-5 are available for immediate download [16, 27]. A user manual on the supercompiler and reports on several interesting experiments with Scp4 can be found in [6–10] (by A. V. Korlyukov). A. P. Konyshev implemented a compiler from the intermediate Scp4's language into the language C [11].

In this paper we continue to describe the supercompiler Scp4 [13, 14]. We give a layout of the supercompiler, after that we make the notations more precisely. Finally, we report a number of experiments with the supercompiler.

Let a program in a language be given, as well as a parameterized input entry of the program. Then such pair defines a partial mapping. By definition, a supercompiler is a transformer of such pairs. The only requirement for the transformer is that it must preserve the mapping values on the mapping domain. The supercompiler Scp4 is a computer

program for transformation of algorithmically defined mappings with the objective of optimization of the running time.

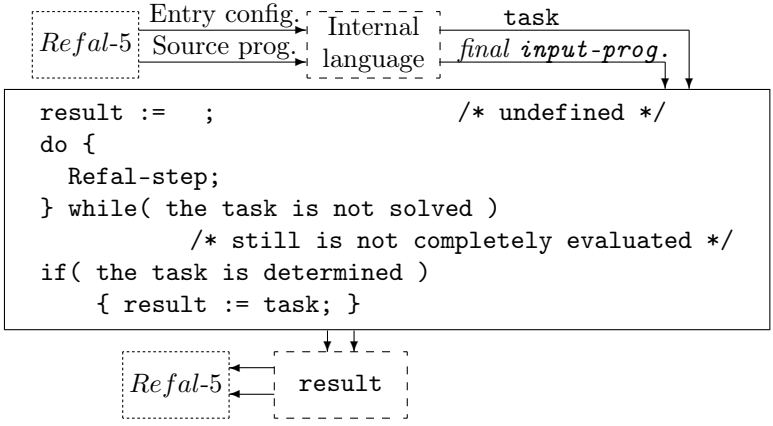
Scp4 unfolds a potentially infinite tree of all possible computations. It reduces in the process the redundancy that could be present in the original program. It folds the tree into a finite graph of states and transformations between possible configurations of the computing system. And finally it analyses global properties of the graph and specializes this graph with respect to these properties (without an additional unfolding). The original program definition is thrown away unchanged, i.e. the resulting definition is constructed solely based on the meta-interpretation of the source program rather than by a step-by-step transformation of the program.

There is a huge amount of literature on program transformation. The closest to our work is the work on generalized partial computation [3], partial evaluation [5, 19], partial deduction [17], deforestation [28] and mixed computation [2]. Both generalized partial computation and supercompilation exploit negative information on the ranges of parameters, on intersection of the ranges, while conventional partial evaluation does not. Both the first two techniques work online, while the third works offline, even though annotation of input programs for Scp4 can optionally be done to improve quality of residual programs. Supercompilation is distinguished from the other techniques by the possibility of extending of the input mapping domain. That allows to use information on function call values, which cannot be completely evaluated during transforming stage. Usage of such kind of information is presented in deforestation, which simplifies the compositional structure of some programs. The idea of the Scp4 general structure is close to the main idea of mixed computation.

The size of the Scp4 system is about 19500 lines of commented pretty-printed source code (800 KB). An online demonstration of the supercompiler is available on an Internet site [16]. All residual programs from our paper were constructed automatically by Scp4 and manually modified for formatting purposes only.

2. Layout of SCP4

This section considers a rough scheme of the structure of Scp4. Given a program written in Refal-5 and a parameterized input entry of the program, Scp4 transforms the pair (into a similar pair in the same language) to decrease its running time on specific input values of the parameters (data) preserving at the same time its running time on all



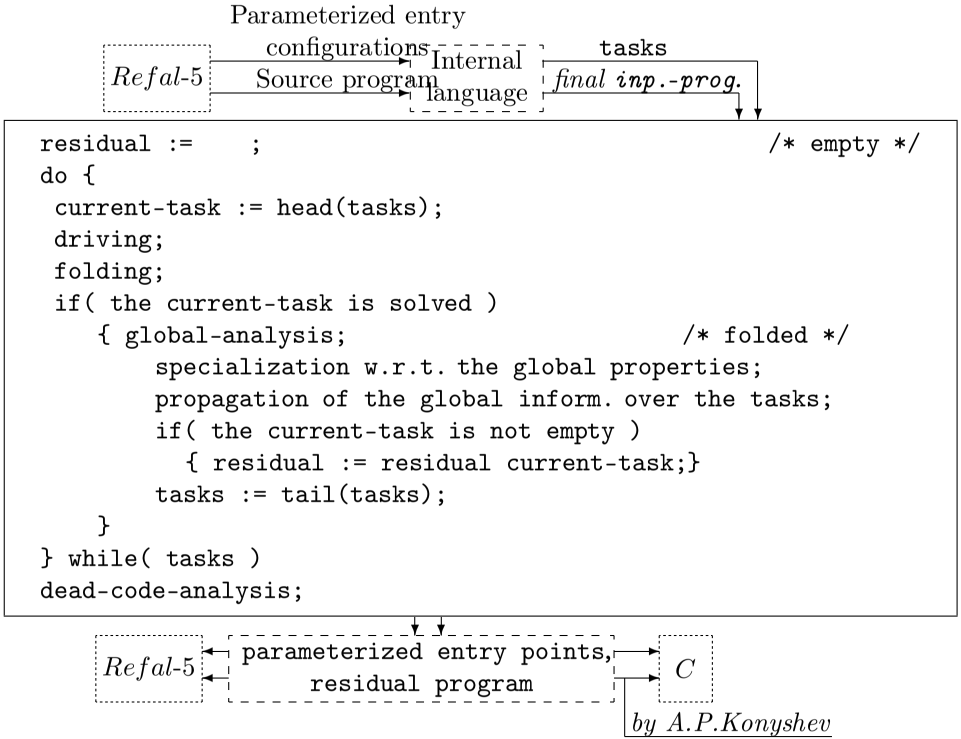
Figures 1. General structure of the Refal interpreter

the other data. By the running time we mean logical time rather than a precise value of any physical clock. The transformer is allowed to extend the domain of the defined mapping one way or another. Operationally, the extension can be expressed as the elimination of both abnormal stops and loops. More precisely, Scp4 receives as an input a number of parameterized entries and it transforms all of the partial mappings at once. We will refer to the entries as the tasks to be solved by Scp4.

We start with an analogy between the Refal-5 interpreter and the supercompiler (see Fig.1 and Fig.2). The analogy reflects the basis of supercompilation. Both the two computing systems work with an internal language, called Refal-graph language. This language describes the transformation (or performing) process, as well as its results, more adequately than Refal itself.

Interpretation consists of elementary logically closed actions (steps). The sequence of steps is completely predetermined by a specific input entry and a program. The result of each step is a new input entry for the subsequent step. Interpretation ends when the task of evaluating of the given entry is fully solved or the interpreter has detected a one-point mapping is undefined on the given input data.

Supercompilation iterates an extension of the interpretation of Refal-graph steps, called *driving*, on parameterized sets of the input entries.



FIGURES 2. General structure of Scp4

Driving constructs a directed tree (a "cluster") of all possible computations for the given parameterized input entry and a given Refal-graph step. The edges of this tree are labeled with predicates over values of the parameters. The predicates specify concrete computation branches. The leaves contain parameterized input entries to the next call of driving. The main aim of driving is to perform as many actions uniformly on the parameterized input data as possible. Immediately after the driving, every Scp4 iteration calls *folding*. The purpose of folding is to split the parameterized entries (to break up a task into subtasks), to transform the whole potential infinite tree (not just a cluster, and, more generally, intermediate potential infinite graph) derived by the driving iterations into a meta-structure – a graph of trees, and, finally, to fold infinite sub-branches of this graph of trees into some finite loops. The subtasks are the roots of the trees of possible computations. We will say a task described in a node is solved if all branches starting in the node are folded. In this case some global properties of the sub-graph rooted in the node are analyzed and the sub-graph is specialized (without any additional unfolding) with respect to the properties. Information on the global properties is propagated over the tasks to be solved. If the current solved task is undefined on all values of the parameters then the branch incoming in the task node is pruned, otherwise the sub-graph outgoing from this node becomes a candidate for a residual "function". The Scp4 iterations stop when the whole potential infinite meta-graph has been folded, otherwise a strategy chooses a new leaf for the following driving. The goal of *dead-code analysis* (see Fig.2) here is to identify input and output formal parameters of the residual "functions", which values do not affect the values of the partial mappings defined with the source tasks (to be supecompiled). In general, dead-code analysis reduces the arities and co-arities of the residual "functions". In particular, removes some function applications. Thus, we emphasize the Scp4's output is defined in terms of the parameters (*semantic objects*).

Further sections describe each process gradually in more details.

3. The REFAL language

The Refal programming language (by V. F. Turchin) is a first-order functional language with an applicative (inside-out) semantics. Roughly speaking, a program in Refal is a term rewriting system. The semantics of Refal is based on pattern matching. As usually, the rewriting rules are ordered to match from the top to the bottom. The terms are generated

using two constructors. The first is the concatenation. It is binary, associative and is used in infix notation, which allows us to drop its parenthesis. In Refal the blank is used to denote the concatenation. The second constructor is unary. It is syntactically denoted with just its parenthesis (that is without a name). Angular brackets are used to denote a function call. Its name is put after the left bracket. Every function is unary. In Refal the ground terms are referred to as constant expressions. Empty sequence is a special basic ground term. This term is denoted with nothing and called "empty expression". It is neutral element (both left and right) of the concatenation. All other basic ground terms are named as "symbols". There exist three types of basic non-ground terms (called variables) - `e.name`, `s.name` and `t.name`. An `e`-variable can take any expression as its value, an `s`-variable can take any symbol as its value, a `t`-variable – any symbol and any expression enclosed in the parenthesis.

Thus, Refal data `d` and patterns `pat` are defined with the following grammar:

```
d    ::= (d1) | d1 d2 | SYMBOL | empty
pat  ::= tpat | pat1 pat2 | empty
tpat ::= (pat) | s.name | t.name | e.name | SYMBOL
```

Here `empty` is the empty string (nihil).

Example: The following program replaces every occurrence of the identifier `Lisp` with the identifier `Refal` in an arbitrary Refal datum.

```
$ENTRY Go { e.inp = <Repl (Lisp Refal) e.inp>; }
Repl { (s.x e.v) = ;
       (s.x e.v) s.x e.inp = e.v <Repl (s.x e.v) e.inp>;
       (s.x e.v) s.y e.inp = s.y <Repl (s.x e.v) e.inp>;
       (s.x e.v) (e.y) e.inp =
            (<Repl (s.x e.v) e.y>) <Repl (s.x e.v) e.inp>; }
```

On the right side of the first sentence of `Repl` we see the empty expression. Below we use sometimes the meta-symbol `[]` for the empty expression. The left sides of the last three sentences and the right side of the second sentence of `Repl` show associativity of the concatenation.

Some additional information about Refal can be found in Appendix A. A detailed description of the language is available in an electronic format [23].

4. Language of parameters

An input entry of a program, called (in Refal) view field, determines the memory part to be evaluated by the abstract Refal-graph machine. One step of the abstract machine is a logically closed transformation of the view field. There naturally appears a sequence:
$ViewField(0) = Entry$;
$ViewField(n+1) = Step(Program, ViewField(n))$;
We are going to give a parameter language for describing of the view fields.

4.1. Parameterized sets of data. At the beginning we define a language for describing some Refal datum sets. It is convenient to represent the language with a pair `positive-pd` and `negative-pd`. The first part represents "positive" information about the data, while the second - "negative". (Below the double brackets are meta-symbols.)

```
pd ::= [[ positive-pd , negative-pd ]]
positive-pd ::= t positive-pd1 | empty
t ::= p | d | (positive-pd)     /* d is a Refal datum */
p ::= s-parameter | e-parameter | t-parameter
s-parameter ::= s.name
t-parameter ::= t.name
e-parameter ::= e.name
empty ::= []                    /* nihil */
```

An `e-parameter` represents the set of all constant expressions (Refal data), an `s-parameter` represents the set of the Refal symbols and a `t-parameter` - the union of the Refal symbol set and the set of all constant expressions enclosed in the parenthesis. Other terms from `positive-pd` are interpreted by the construction of the terms.

Below parameters are named by natural numbers, while variables are named by identifiers (to exclude confusion).

```
negative-pd ::= restriction, negative-pd | [] | ∅
restriction ::= st1 # st2 | e-parameter # []
st ::= s-parameter | SYMBOL
```

`s.n1 # s.n2` is a restriction on the sets of the parameter ranges: the sets do not intersect. `s.n # SYMBOL` and `SYMBOL # s.n` are restrictions on the set of the parameter range: `SYMBOL` does not belong to the set. `SYMBOL1 # SYMBOL2` represents the empty set if these symbols do not coincide and is a tautology otherwise. `e.n # []` - the empty expression does not belong to the parameter range. The empty expression `[]` staying

alone represents a tautology, while ∅ - the empty set. Comma (here and in the pair defining `pd`) is interpreted as a sign for intersection of the sets.

4.2. Parameterized sets of view fields (stacks) and Refal - expressions. The parameter language used by Scp4 is a first order language: there are no parameters, which domains contain function names or function application constructors. All function names represent themselves. The constructor of function application `< ... >` is a constant encoded with the data `(Call` *INFO*`)`. *INFO* is an information about the call, in particular, it contains description of the domains of the function call arguments and the image of the call written in the `positive-pd` language. Graphically coinciding parameters from the arguments of different calls represent the same set. The links between the elements of the stack are described by formal output variables `out.name`, which are only fixed meta-names for the values of the corresponding function calls. The "negative" part of the description of the parameters is common for all calls from the stack.

Example: `<F s.1>←out.1; <G <F e.2> out.1>←out.0;`
Here the sign ← means an assignment of the left side to the right[1] one. Sometimes we will write `<F e.2>←e.4` and mean that the domain of the right side is restricted by the `F`'s image.

5. Internal language

Henceforth, we assume all trees are developed from the left to the right. The internal working language of the transformations is a language, to which V. F. Turchin refers as Refal-graphs. This language describes the transformation process, as well as its results, more adequately than Refal itself. Here, we consider some properties of the Refal-graph language. Our purpose is to introduce the reader the main concepts. A detailed description of the language may be found in [13]. See also [21].

[1] We use the Turchin's arrow denotations [24–26] for describing the process of supercompiling. The meaning of the arrows is "the range of the part originating an arrow is narrowed to the range of the other part where the arrow incomes".

5.1. Input subset of the Refal-graph language. The input subset is an applicative functional language based on pattern matching with syntactical means for describing the directed trees for analysis of all patterns (from a "function") at once. Nodes of the trees are "functional", i.e. without backtracking, edges are numbered. The patterns are explicitly decomposed using a finite set of simple patterns (Sect. 6.1). Some negative information reflecting failure over the upper branches (in a given branching) and expressible in the language `negative-pd` (where the parameters are treated as the variables) is written explicitly (on the given edge) in `negative-pd`. Leaves of the trees correspond to the right sides of the Refal-sentences, hence the concept of a step of the abstract machine of the input subset of the Refal-graph language is determined (straightforward similar to the basic Refal step: choosing of an active function call, pattern matching and replacement of the active call with the right side of the sentence chosen by the pattern matching). Every function from this input subset is unary.

Example: The following two definitions specify a function `F` in Refal and in the input subset of Refal-graph language.

```
F {
A e.x   = e.x;
s.y e.x = s.y e.x;
        = ;
}
```

```
F {
+[1] e.inp→s.v e.u;
 :{ +[1] s.v→A;{e.u←e.out;};
    +[2] s.v # A; {s.v e.u←e.out;}; }
+[2] e.inp→[]; {[]←e.out;};
}
```

Where the arrow → is interpreted as matching of the variable's value from the left side with the pattern from the right side (of the arrow). See Sect. 4.2 for the meaning of the ←. The digits inside the square brackets are numbers of the branches in a concrete branching. The plus sign stands for an alternative.

5.2. Internal language for transformations. In effect, programs written in the input fragment of the internal language are not subjects to be transformed by Scp4, but only to be performed. Scp4 derives new programs in the internal transformation language (the whole Refal-graph language). The transformation language is a superset of the input Refal-graph language.

The colored graph nodes contain parameterized description of the function stacks (Sect. 4.2) at the given points. (We will explain the colorings later.) The nodes also may contain global information of the supercompiling process. The functionality of the nodes remains valid.

Every node has a unique name, which is the creation time of this node. There is no analogue for the Refal division of the sentences into the two sides. A function application may appear on an edge and the application's value may be tested along the edge likewise the input arguments to the graph. Thus this language allows decomposition of the function calls onto single edges. The decomposition may be done by means of `Let`-constructors. "Functions" in this language may have nontrivial arities and co-arities. A "function" transforms the environment defined by the input format and the active call. The function result is an environment defined by the output format and the substitution at the end of the chosen branch completes the recursion. Here the analogue of the Refal-step is a sequence of the elementary actions starting with an active function call and ending with searching of the next active call. The output function formats and the function calls are colored.

Example:

```
* Input Format:  { e.1←e.1; }
F7 {
 +[1] e.1→[]; (I)←e.out;
 +[2] e.1→I e.11; {e.11←e.1;} <F7 e.1> {e.out←e.2;}
      e.2→(e.3); {e.11←e.1; e.3←e.4;}
      <F16 e.1, e.4> {e.out1←e.5;} {(e.5)←e.out;}
}
* Inductive Output Format:  { (e.6)←e.out; }
```

That is a fragment of a definition written in the internal language. The output and input environments of the function call are explicitly written. We see the two "steps" on the second branch. The `I` is a constant.

6. Driving

Consider a school algorithm (given in a non-formal language) for calculation of the root set of a liner equation with one variable. Let us think of coefficients of the monomials as data and of the other part of the syntactical equation structure as a program. Thus, our algorithm is turned into an "interpreter" - a solver for the program-equation. This solver has no loops (we assume arithmetic operations are basic primitives). Next, consider the liner equations with parameters: some of the data-coefficients became parameters. A schoolboy solving such equations is a master of the driving concept. The answer, naturally, is expressed

in the parameter terms and is a tree splitting the parameters and performing algebraic actions being uniform on the parameters. Recall that above we named the result of the driving as cluster.

Chosen a parameter language we are able to refine upon the concept of the driving in the input subset of the Refal-graph language. This section considers just this subset. First, we are going to focus on the strict driving (corresponding to the strict Refal-graph interpretation), next we will give some remarks on the lazy driving.

6.1. General structure of driving. The driving is an extension of the interpretation of *one* Refal-graph (its input subset) step on the parameterized sets of the input entries. The purpose is to perform as many actions of the Refal-graph machine uniformly on the parameter values as possible.

A step of the abstract Refal-graph machine:

`Step: prog` × `name` × `D-env` ↦ `stack` × `D-env, D-env: vars` ↦ `data`

consists of two logical stages. On the first stage the machine chooses a path from the root of a graph to a leaf, on the second stage the function stack is modified according to the leaf syntax and the environment calculated on the first stage. In fact there exists a single variable in the environment, but we prefer to think about the variable as a set of the variables consisting of one variable.

Refine on the interpretation. On the first stage, the values of the variables from the current environment match the simple patterns (Sect. 5.1). (The task for matching a datum `d` to a patter `p` is a task for solving an equation `p` = `d`. The result of the solution is either the values of the variables from the pattern `p` or the information that the equation has no solutions.)

- If the matching is successful, then the environment is modified. If an edge is passed: the machine reached either a branching-point or a leaf. In the second case, the path is chosen. In the first, the current environment is stored in the branching point (a node) and the machine passes to the consecutive matchings of the current values of the variables with the patterns labeling the first edge outgoing from the current node.

- If the matching is fail, then the backtracking to the closest branching-point happens, where the environment (saved in this node before) is restored, and the next branch outgoing from the node is considered as a possible path for reaching a leaf.
- If the closest branching is exhausted, then the partial mapping corresponding to the graph is declared as undetermined on the given input environment.

Remark 1: Functionality (Sect. 5.1) of the Refal-graph branching-point implies every node in the graph, for the sake of interpretation, is equal in its properties to the graph entry point: the pair `(Graph-name, Node-identifier)` can be considered as an independent "function"-subgraph. The arity of the subgraph is defined by the number of the variables in the environment `env` of the given node. Henceforth, we will denote the entry point to the subgraph by $\texttt{<Graph-name}_{nd-id}\ \texttt{env>}$.

$\texttt{Driving: prog} \times \texttt{name}_{nd-id} \times \texttt{PD-env} \mapsto \texttt{Cluster}, \texttt{PD-env: vars} \mapsto \texttt{pd}$

Driving receives as an input a parameterized environment `Pd-env`, the values of the variables in which there may be an arbitrary parameterized set of the data, in accordance with the variable's type (Sect. 4.1). When `Driving` reaches a branching-point, like `Step`, it stores the current state of the parameterized environment (in this node), takes the first edge outgoing from the node and tries consecutively to solve the equations with parameters $\texttt{p} = \texttt{pd}$, where the `p`-s are patterns labeling this edge and the `pd`-s are the values of the parameterized variables to be matched.

Remark 2: By definition, the supercompiler may extend the domain of the partial mapping defined by the parameterized entry point and the program to be transformed. That provides a possibility to use lazy evaluation in supercompile-time. I.e. the driving may be an extension of one step of a lazy abstract Refal-graph machine. In this case, the variables in the parameterized environments take any parameterized Refal expressions (Sect. 4.2), according to the variables' ranges. The user may choose a concrete strategy for the driving (strict or lazy).

Definition 1: The syntactical construction `variable→pattern` is called *contraction*. Semantics: the value of the variable has to match the pattern. A `contraction` is called elementary, if and only if

$\texttt{contraction} \in \{\ \texttt{e.n}\rightarrow\texttt{(e.n}_1\texttt{)}\ \texttt{e.n}_2,\ \texttt{e.n}\rightarrow\texttt{e.n}_1\ \texttt{(e.n}_2\texttt{)},\ \texttt{t.n}\rightarrow\texttt{(e.n}_1\texttt{)},$
$\texttt{e.n}\rightarrow\texttt{SYMBOL e.n}_1,\ \texttt{e.n}\rightarrow\texttt{e.n}_1\ \texttt{SYMBOL},\ \texttt{t.n}\rightarrow\texttt{SYMBOL},\ \texttt{s.n}\rightarrow\texttt{SYMBOL},$
$\texttt{e.n}\rightarrow\texttt{s.n}_1\ \texttt{e.n}_2,\ \texttt{e.n}\rightarrow\texttt{e.n}_1\ \texttt{s.n}_2,\ \texttt{t.n}\rightarrow\texttt{s.n}_1,\ \texttt{e.n}\rightarrow\texttt{t.n}_1\ \texttt{e.n}_2,$

`e.n→e.n1 t.n2, s.n→s.n1, e.n→[] }`.

The patterns from the right sides of the elementary contractions are called elementary patterns.

Statement 1. *There exists an algorithm for "solving" any equation* ***p*** = ***pexpr****, where* ***p*** *is an arbitrary elementary pattern and* ***pexpr*** *is an arbitrary parameterized Refal expression (in our language of parameters, see Sect. 4.2). The result of the algorithm is:*

- *the values of the pattern variables written in the language of the parameterized Refal expressions;*
- *or a directed finite tree (cluster) splitting the equation parameters in the sub-cases (subsets) "*`if P(...) then ...else if Q(...) ...`*", which predicates are written in the elementary contraction language and which leaves are the descriptions of the pattern variables' values (from the right sides of these contractions) written in the language of the parameterized Refal expressions;*
- *or the information that the root set of the equation (for all values of the parameters) is the empty set;*
- *or a parameterized function call such that the equation solutions depend on the value of the call.*

The algorithm for solving the parameterized equations may be found in [21,24].

Corollary 1. *Predicates labeling the driving cluster edges can be written as compositions of the elementary contractions on the parameters.*

Recall that natural numbers name the parameters, while identifiers name the variables.

Example 1:

The equation: `s.x e.y = [[s.3 A e.2 <F e.1>, s.3 # B, e.1 # [] ]]`

The solution has no additional conditions on the parameters:

`[[{s.x = s.3, e.y = A e.2 <F e.1>}, s.3 # B, e.1 # [] ]]`

Example 2:

The equation: `s.x e.y = [[ e.1 A e.2 <F e.1> , e.2 # [] ]]`

The solution:

```
if e.1→[]; then [[ {s.x = A, e.y = e.2 <F >}, e.2 # [] ]]
else if e.1→s.11 e.12;
  then [[{ s.x = s.11, e.y = e.12 A e.2 <F s.11 e.12>}, e.2 # [] ]]
       e.y = e.12 A e.2 <F s.11 e.12> }, e.2 # [] ]]
```

```
else "No roots";
```

Example 3:
The following equation can arise from the composition of the following three elementary contractions `e.n→s.x e.m; e.m→s.z e.y; s.z→s.x;`.
`s.x s.x e.y = [[ s.3 s.4 e.2 <F e.1> , s.4 # A ]]`
The solution:

```
if s.4→s.3; then [[{s.x = s.3, e.y = e.2 <F e.1>}, s.3 # A ]]
else "No roots";
```

Example 4:
The equation: `e.y s.x = [[s.3 A e.2 <F e.1>, e.2 # [], s.3 # A]]`
The solution depends on the values of the call `<F e.1>`.

Dividing of the parameters into the subsets depends on a concrete algorithm solving the parameterized equations. Further actions of the driving depend on the results of the algorithm.

Remark 3: The number of edges outgoing from a node of the cluster may be grater, lesser or equal to the number of the edges outgoing from its pro-image (under the mapping of the driving).

Remark 4: A contraction on an edge of the cluster may be derived, which is absent in the source graph.

Remark 5: There can be constructed a node such that all edges outgoing from the node have a common contraction.

Example 5: (of driving $a \times x + b = 0$)

```
$ENTRY Go {e.1 s.2 (e.3) = <Linear e.1 s.2 e.3>;}
Linear { 0   0  = "Any real number";
         0  s.b = "No roots";
        s.a s.b = (Div ('-'s.b) s.a); }
```

The following pair is input to the driving. '**:{**' stands for the branching point.

```
{ e.1 s.2 e.10←e.inp; };
Linear {
+[1] e.inp→s.u2 e.x; e.x→s.u3 e.x1; e.x1→[];
   :{+[1] s.u2→0;
       :{+[1] s.u3→0; {"Any real number"←e.out;};
         +[2] s.u3 # 0; {"No roots"←e.out;};
        };
    +[2] {(Div ('-' s.u3) s.u2)←e.out;};
   };
}
```

The cluster of the driving:

```
e.1→e.1 s.2 (e.10);
:{+[1] e.1→s.3 e.11; e.11→[]; e.10→[];
  :{+[1] s.3→0;
    :{+[1] s.2→0; {"Any real number"←e.out;};
     +[2] {"No roots"←e.out;};
    };
   +[2] {(Div ('-' s.2) s.3)←e.out;};
  };
  +[2] e.1→[]; e.10→s.5 e.11; e.11→[];
    :{+[1] s.2→0;
      :{+[1] s.5→0; {"Any real number"←e.out;};
       +[2] {"No roots"←e.out;};
      };
     +[2] {(Div ('-' s.5) s.2)←e.out;};
    };
};
```

6.1.1. *Reconstruction of the function stack.* In the case of the lazy driving, a function call <$\mathtt{F}_{nd-id}$ env> may be a hindrance for solving a parameterized equation (see Statement 1), i.e. it is necessary to know some properties of the call value. Then the function stack is reconstructed - the call <$\mathtt{F}_{nd-id}$ env> is pushed out on the top of the stack in the closest branching-point, and afterwards the driving works with the definition $\mathtt{F}_{nd-id}$. The stop point of the driving of the current call <$\mathtt{Current}_{nd-id1}$ env1> (over the Refal graph) is memorized in order, if needed, to continue the driving of <$\mathtt{Current}_{nd-id1}$ env1> from this point.

Corollary 2. *A cluster of the driving may contain the sub-trees corresponding to diverse subgraph-"functions" of the program to be transformed.*

Remark 6: The driving can start with any node of the source graph rather than only with the root of the graph.

Statement 2. *The partial mapping specified by the driving is a total mapping. (No infinite loops).*

The statement is implied by the two remarks: 1) the number of reconstructions of a given stack is bounded by the number of function calls in the stack; 2) the number of nodes in the source graph is finite.

6.1.2. *The strategy for deriving of an input format.* The nodes of the driving cluster are painted in two colors distinguishing the *pivot* and the *secondary* points. In Scp4 we use the following strategy to paint cluster nodes.

If all leaves contain trivial stacks (no function call), then all branching-points are marked as secondary, otherwise

- the left-most branching-point of the parameters is marked as pivot, all others are painted in the secondary color;
- it is possible that the root of the cluster is not a branching-point; in this case we paint it as secondary when there exist branching points in the cluster. If there are no branching-points, then the Refal-step can be uniformly interpreted on the parameterized input data. In the last case a color of the root depends on the strategy of the driving chosen by the user.

7. Folding

The second tool is `folding`. The tool has no analogies in the abstract Refal-graph machine. (In the case of "utterly lazy" evaluation, the search for the call to be evaluated among the previous completely evaluated calls may be considered as an analogue.) It folds the meta-tree of all possible computations in a finite graph and is launched immediately after the driving. This tool takes into account only the pivot nodes. Exactly those of them, which really fold the tree, are declared as basic.

Refine: the folding tries to wrap the part of the path, which is not folded to a given moment, from the meta-tree root to a pivot node (Sect. 6.1.2), further referred to as `Current-node`, in the current driving cluster if such node does exist. The folding is divided in two logically closed tools: reducing and generalization. Both tools use not only syntactical but also semantic properties of the function stack. For the first time the idea of the semantic part of the algorithm was reported in Obninsk (1989) by Valentin F. Turchin [22].

7.1. Reducing. Recall that we are working with the first order language of the parameters. Let the function call stack grow from the right to the left with the left most call active. Let a function call stack be split into a starting non-zero length segment and the tail part, then the computations on the stack are consequently performed by the starting segment and, afterwards, by the tail in the context of the environment calculated during the first stage.

Given a current pivot node, the reducing looks for another pivot node (called `Previous-Node`) on the above-mentioned path, such that the set of function stacks (described by the parameters) of the node not only is a superset of the set of function stacks of a starting segment of the current node, but also there exists a parameter substitution that reduces the description of the previous set of stacks to the description of this starting segment of the current node. Thus, all computations defined by this starting segment, by means of the substitution (maybe defined indirectly through a function call), can be reduced to the computations defined by the previous pivot node. The tail part of the current parameterized stack is declared as a separate task for the supercompiler: the result of evaluation of the starting segment is declared as unknown at this stage of the transformations. The output environment of the segment is completely undefined, i.e. it is described by some fresh `e`-parameters.

Example 1: Consider the following two stacks:
`[[<F t.10 e.11>, ]]` and `[[<F e.1 t.2>, ]]`. The sets defined by these parameterized descriptions coincide, but there are no substitutions of the parameters reducing one description to the other.

If there is no suitable previous pivot node, then the meta-graph of all possible computations remains unchanged, otherwise, the current stack is split into two: the first part of the task (defined in the current stack) for developing the meta-graph is reduced to the task considered before (see above). An induction step on the structure describing the function stack was exercised. Besides, both the splitting of the stack and the reducing substitution define a set of sub-tasks and relations between them: a decomposition of the tasks has occurred. The decomposition forms a sequence of the tasks for developing the trees of possible computations. Further the tasks are resolved one by one. From the point of view of the supercompiler, these tasks are the entry points for transformations going on in the context of the meta-graph constructed by the given moment.

7.1.1. *The strategy for processing of the meta-tree.* As mentioned above the subject to be folded is the path from the meta-tree root to the current node. The pivot nodes are run consequently along this path according to its orientation - from the left to the right (in other words, in the order of the time of creation of the nodes). Before every attempt to reduce the current pivot node to the previous pivot node, the reducing algorithm tries to reduce the current node to the basic nodes of the completely folded sub-trees, which roots are located at the ends of the edges outgoing from the previous node and different from the branch incoming

into the current pivot node. The basic nodes are considered also in order of time of creation of these nodes.

The parts of the meta-graph, which are under the basic nodes, are called the components of folding. The components are the candidates for inclusion into the residual graph.

7.2. Generalization. From the point of view of generalization, the set of parameterized stacks of the pivot nodes from the partially folded (potentially infinite) meta-tree is a set of the induction hypotheses of every path in the tree will have a pivot node in a cluster of the driving such that the stack of the node will be completely reduced to the stacks of the other basic or pivot nodes existing in the meta-tree at the moment of the reducing attempt.

The algorithm of generalization is a tool for guaranteeing of finiteness of the number of these hypotheses, which succeeded in "proving" during development of the tree. If the reducing has not found a pivot node to which the current node is reduced (maybe partially), then generalization looks for a previous pivot node (on the path from the tree root to the current pivot node), which is "similar" to the current pivot node.

7.2.1. *Similarity.* A "similarity" relation links semantic loops in the meta-tree with syntactical structures in this tree. It is convenient to render the relation as intersection of a "similarity" relation of a pure function structure of the stacks (of a sequence of the function names) and "similarity" between the parameterized descriptions of the arguments of the different calls for the same functions. At the beginning, we are going to pay attention only to the function call names, to be more precise — to the call names syntactically being the uppermost calls (w.r.t. the composition structure) in each stack element. The following condition for approximation of a loop in the meta-tree of all potential computations was suggested by V. F. Turchin in Obninsk (1989,[22]).

Let each function call for `F` be attributed by the time ($\mathtt{F}_{time}$) the call was created during development of the meta-graph. Given two stacks labeled by the times - `Previous` and `Current`, we say that this pair indicates a loop if the stacks can be represented in the form:

```
Previous = PrevTop; Context;
Current = CurrTop; Middle; Context;
```

where the bottom of the stacks is the length-most common part (maybe which length is zero), i.e. this part nowise took part in the process developing of the function stack along the path from the previous node to

the current, and the "tops" of the two stacks
(`PrevTop = F1`$_{ptime_1}$`, ..., FN`$_{ptime_n}$
`CurrTop = F1`$_{ctime_1}$`, ..., FN`$_{ctime_n}$) coincide modulo the creation times of the calls. The top `PrevTop` is called the entry point of the loop. The body of the loop is defined by the path (from the previous pivot node to the current), that accumulates the `Middle` of the stack. The `Context` determines computations immediately after the loop.

The definition reflects an assumed development of the stack as follows `PrevTop; Middle`^{n}`; Context` if the `Middle` and `PrevTop` are considered modulo the creation times of the calls. In this case, if "similarity" of the considered nodes will be confirmed by other criteria (see below), then the branches outgoing from the previous node are pruned and the task described in the node is split into the subtasks corresponding to the loop and the context.

Example 1: Consider a definition of the unary factorial.

```
$ENTRY IncFact {e.n = <Plus (I) (<Fact (e.n)>)>;}
Fact { () = I;
       (e.n) = <Times (e.n) (<Fact (<Minus (e.n) (I)>)>)>; }
Times { (e.u) () = ;
        (e.u) (I e.v) = <Plus (<Times (e.u) (e.v)>) (e.u)>; }
Plus { (e.u) () = e.u;
       (e.u) (I e.v) = I <Plus (e.u) (e.v)>; }
Minus { (I e.u) (I e.v) = <Minus (e.u) (e.v)>;
        (e.u) () = e.u; }
```

"Lazy" development of the stack along the main branch yields the following sequence:

[1]: `IncFact`$_1$`;`
[2]: `Plus`$_2$`;`
[3]: `Fact`$_4$`; Plus`$_3$`;`
[4]: `Times`$_5$`; Plus`$_3$`;`
[5]: `Fact`$_7$`; Times`$_6$`; Plus`$_3$`;`
...

Here the Obninsk condition indicates that the stack formed on the fifth step is "similar" to the stack created on the third step. A hypothesis is that the stack will further have the form `Fact; Times`^{n}`; Plus`$_3$`;` , where `Fact; Times`^{n}`;` is the body of the loop, while `Plus`$_3$ represents computations immediately following the loop. Note the arguments of the calls for `Plus`$_3$`, Plus`$_2$`, Times`$_5$`, Fact`$_7$ contain other calls.

Definition 1: Let a finite alphabet $A ::= \{ \mathtt{a}_i \}$ be given. Let W stand for the set of all finite words over A including the empty word. Let a function `G`: $A \mapsto W$ and a word $\mathtt{s} \in W$ be given. Define a sequence (possibly finite) $\mathtt{stack}_n$ fixing the birth-times of the calls:
1. $\mathtt{stack}_0$ `::= (s,0)`
2. defined $\mathtt{stack}_i$ `::= (`$\mathtt{a}_i$`, t)` $\mathtt{u}_i$, where $\mathtt{a}_i \in A$, $\mathtt{t} \in \mathbb{N}$ and $\mathtt{u}_i$ is the tail of the sequence, then $\mathtt{stack}_{i+1}$ `::= Time(G(`$\mathtt{a}_i$`), MaxTime)` $\mathtt{u}_i$, where `MaxTime` is the maximum on the second components of the pairs from $\mathtt{stack}_i$.

```
Time(a w, time) ::= (a,time+1) Time(w, time+1);
Time( , time) ::= ;                          ,where a ∈ A, w ∈ W.
```

The sequence $\mathtt{stack}_n$ is called the timed development of the stack `(G,s)`.

Statement 1. *(V. F. Turchin [22]) Any development of a timed stack* $\mathtt{stack}_n$ *is either finite or there exist natural numbers* k *,* m *such that* $k > m$ *and* $\mathtt{stack}_m$ `= (`a_1`, `t_{m1}`) ...(`a_i`, `t_{mi}`) context`
$\mathtt{stack}_k$ `= (`a_1`, `t_{k1}`) ...(`a_i`, `t_{ki}`) middle context`
where `middle` *and* `context` *are sequences of the pairs of the form* `(`a_j`, `t_j`)`*, possibly having zero lengths.*

Henceforth, the part of the stack `(`$\mathtt{a}_1$`, `$\mathtt{t}_{m1}$`) ...(`$\mathtt{a}_i$`, `$\mathtt{t}_{mi}$`)` responsible for the entry point of the loop will be called prefix.

Every pivot node in the meta-tree of all possible computations has a node - pro-image in the source graph. Every node in the input graph has a unique name. Scp4 considers the set of the names $\mathtt{FunctionName}_{nd-id}$ stored in the nodes by the driving and the stack's reconstruction (Sect. 6.1) as an alphabet examined by the Obninsk condition. The set of these names is finite.

$\mathtt{FunctionName}_{nd-id}$ completely specifies the structure of the environment - the variables, which values determine evaluation of this call. Therefore, we have:

Statement 2. *Let* `(`a_1`, `t_{m1}`) ...(`a_i`, `t_{mi}`)` *and* `(`a_1`, `t_{k1}`) ...(`a_i`, `t_{ki}`)` *be the prefixes indicated by the Obninsk condition, then for every* j $(1 \leqslant j \leqslant i)$ *the arities of the environments and the names of the environment variables coincide.*

After that, a simplification ordering condition specifies a "similarity" relation on the parts of the parameterized environments `(`$\mathtt{a}_j$`, `$\mathtt{t}_{mj}$`)`, `(`$\mathtt{a}_j$`, `$\mathtt{t}_{kj}$`)`, which provides "positive" information of data (Sect. 4). This relation is a variant of a relation originating by Higman [4] and Kruskal

[12] and is a specification of the following term relation. Let two terms `Previous` and `Current` be written on a blackboard by a chalk. We say the term `Current` is not less complex compared to the `Previous` iff the `Previous` can be obtained from the `Current` by erasing some basic terms and constructors (see also [20, 25]). If the set of the basic terms is reasonable enough, then any infinite term sequence $\mathtt{t}_n \in$ `positive-pd` has a pair $\mathtt{t}_i$, $\mathtt{t}_k$ such that $k > i$ and $\mathtt{t}_k$ is not less complex compared to $\mathtt{t}_i$ [4, 12].

Scp4 uses also other additional "similarity" conditions. Some of them reflect technical details of the implementation, others were taken *ad hoc* based on the experience of the authors.

Our approximation of the concept of the loop is the assumption that this loop constructs the terms and constructors erased by us (see above). Both prefixes are entry points of the loop: the previous is the first iteration, the current is the second one.

To fold the path in the meta-tree, we have to reduce (by a substitution of the parameters) the parameterized description of the environment of the current prefix to the environment of the previous prefix. If there are no such substitutions, then the descriptions of these two environments are subjects for generalization. I.e. a new parameterized environment is constructed such the both prefix environments can be reduced to the new environment by some substitutions of the parameters. The previous prefix is replaced with the generalized prefix.

After the generalization of the `positive-pd` parts, it is necessary to generalize the `negative-pd` parts of the descriptions. The result of the generalization has to be again in the `negative-pd` language. We refer the reader to the sources of Scp4 [16] for the details.

7.2.2. *The strategy for processing of the meta-tree.* Generalization processes the pivot nodes along the "path" from the current node to the meta-tree root, i.e. in the opposite direction to the edge orientations - from the right to the left. Let us formulate several properties of the generalization tools.

Statement 1. *Let* $\mathtt{Previous} \propto \mathtt{Current}$ *stand for for the Obninsk relation for indicating the loop (Sect. 7.2.1). Given three timed parameterized stacks* $\mathtt{Stack}_{t_1}$, $\mathtt{Stack}_{t_2}$, $\mathtt{Stack}_{t_3}$ *on the path from the meta-tree root to the current node, where* $t_1 < t_2 < t_3$, $\mathtt{Stack}_{t_1} \propto \mathtt{Stack}_{t_3}$ *and* $\mathtt{Stack}_{t_2} \propto \mathtt{Stack}_{t_3}$. *Then 1)* $\mathtt{Stack}_{t_1} \propto \mathtt{Stack}_{t_2}$; *2) the lengths of the prefixes* $\mathtt{Prefix}_{t_1}$, $\mathtt{PrefixA}_{t_3}$ *indicated by the relation* $\mathtt{Stack}_{t_1} \propto \mathtt{Stack}_{t_3}$ *are*

not grater than the lengths of the prefixes $\mathtt{Prefix}_{t_2}$, $\mathtt{PrefixB}_{t_3}$ *indicated by the relation* $\mathtt{Stack}_{t_2} \propto \mathtt{Stack}_{t_3}$; *3) the length of the context* $\mathtt{Context}_{t_1}$ *indicated by the relation* $\mathtt{Stack}_{t_1} \propto \mathtt{Stack}_{t_3}$ *is not grater than the length of the context* $\mathtt{Context}_{t_2}$ *indicated by the relation* $\mathtt{Stack}_{t_2} \propto \mathtt{Stack}_{t_3}$.

Therefore, under the given strategy the Obninsk relation selects the loop with the least length and the entry point in the loop is represented by the largest function stack among the prefixes of the stacks being in this relation with the current stack. That provides possibilities for optimization of this composition (of the prefix and the context) inside this loop. On the other hand, the largest context reflects the semantics of the computations, which, are irrelevant to this loop and this context will be unfolded-folded separately.

8. Unfolding

The abstract Refal-graph machine, immediately after the current step, modifies the function stack - prepares its syntactical structure for the next step. Choosing of the active function call during interpretation is an analogue of unfolding.

The abstract unfolding: a) reconstructs the structures of the parameterized function stacks in the leaves of the driving cluster in accordance with a strategy (see below); b) chooses (in the meta-tree) a leaf, which stack will be transformed by the subsequent call for the driving. In our transformer the strategy chooses the leaf of the top-most branch among the branches having at least one function call on their ends. I.e. the strategy is the depth-first strategy.

Recall that the stack extends from the right to the left. Another strategy reconstructs the description of the parameterized stack. After the driving, each element of the stack is represented by a parameterized expression (roughly speaking). The following two strategies for development of the stack are implemented: a) the lazy strategy corresponds to the lazy (call by need) semantics of functional languages (the given stack development during supercompilation does not contradict the Refal semantics, because it can just extend the domain of the partial mapping); b) the applicative strategy corresponds to the applicative semantics (call by value). Constructors from the top level of the stack elements, i.e. the constructors that are not pieces of the function call arguments, are moved to the right side along the stack. The right-most stack element

(the bottom) is a common context for all computations in the concrete task and is an accumulator for the constructors.

Example 1: The stack
`[[ A <G B <F e.1»←out.1; <H e.1 out.1>←out.2; , ]]`
will be transformed by the lazy strategy to the form
`[[ <G B <F e.1»←out.3; <H e.1 A out.3>←out.2; , ]]`,
while by the applicative strategy to
`[[ <F e.1>←out.3; <G B out.3>←out.4; <H e.1 A out.4>←out.2; , ]]`.

9. Global analysis

Suppose, a pivot node was declared as basic during the transformations and the sub-tree outgoing from the node is completely folded. We refer to this node as the input point of the folded-component. An analysis of global properties of this folded-component starts.

Three different kinds of components can appear in the process: 1) self-sufficient, i.e. those of them which refer just to themselves; 2) components that refer to themselves and to the components constructed before; 3) components that contain references to the basic nodes inside the part of the tree on which folding has not been finished yet. This classification is defined for a particular moment of the folding procedure, and it depends on this algorithm.

Some properties of the components are more suitable to be analyzed in terms of the Refal-graph language others - in terms of the Refal language.

From the point of view of an input program to be transformed the properties analyzed by Scp4 are quite trivial and rather uninteresting, but the given subgraph-component was constructed automatically. This is the result of specialization of the program and simplification of its compositional structure. Any trivial transformations are very desirable at this moment of supercompilation, because further transformations of still unfolded parts of the tree of possible computations depend on their results. The principal point is to discover an output format of the folded-component; breaking up a task into subtasks during supercompilation leads to the total loss of information about the images of the subtasks (Sect. 7). An inductive output format is a common kind of the structure of the subtask image. After the global analysis this information is propagated through connections between the tasks and is used later.

The global analysis and transformations on the basis of the properties found are logically separated one from the others. The transformations extend the domain of the partial mapping defined by the Scp4 input and are able to decrease significantly time complexity of the programs to be transformed. The details of the global transformation can be found in [13, 14, 16].

9.1. Analysis in terms of the Refal-graphs. The following properties are analyzed in terms of the Refal-graph language. A sub-graph (a folded component) is said to be empty iff no leaves of the sub-graph can be reached on any input data. The syntactical emptiness of the sub-graph is analyzed and the edges incoming to the empty sub-graph calls are pruned in the meta-graph. Hence, new uniformly interpreted steps may appear in the meta-graph. Let the image of a partial function be the only point. In this case, we can replace each call of the function with its value. Such syntactical constant sub-graphs are looked for.

As we mentioned above the output formats of the folded components are derived. Derivation of the output formats is critical when the length of the function stack of a program to be transformed is not uniformly bounded on the input data. Without such derivation interesting transformations are not happening. The output formats and the function (sub-graph) calls are painted in two colors. An output format is declared inductive when it has been just constructed during transformations (the term takes its origin from the method that constructs the output formats: an inductive hypothesis and its automatic proof); a graph call is declared inductive immediately after use (creation) of its real output parameters via the inductive output format of the graph. The tool creating the inductive hypothesis is the algorithm for generalization of Refal-expressions defining the output structures on each branch of the sub-graph [13].

Example 1: Let the following Refal program be an input to Scp4. `$ENTRY Go { e.n = <Plus (I I I) (e.n)>; }`, where the `Plus` is defined in Sect. 7.2.1. The definition (viewed separately from the context of the function call) does not give any information about the output structure of `Plus`. At the stage of constructing of its output format, after the specialization w.r.t. the call context, the folded component looks as follows.

```
(Input-Format e.1←e.1;)
F7 {
+[1] e.1→[]; I I I←e.out;
+[2] e.1→I e.2; {e.2←e.1;} <F7 e.1> {e.out←e.3;}; I e.3←e.out;
```

```
} (Output-Format e.out←e.out;)
```

It is possible to construct a non-trivial output format. Our algorithm constructs the inductive output format `I e.4`, though a more precise analysis is able to give the following perfect variant `I I I e.4`. The description of the subgraph `F7` is made more precise (the double brackets are the inductive colors):

```
(Input-Format e.1←e.1;)
F7 {
+ e.1→I e.2; {e.2←e.1;} «F7 e.1»{{I e.out1←e.3;}}; I e.3←e.out;
+ e.1→[]; I I I←e.out;
} (Inductive-Output-Format I e.out1←e.out;)
```

9.2. Analysis in terms of the Refal Language. This analysis allows to transform some simple recursive definitions to one-step programs. Here we just give two examples and refer the reader to [14].

Example 1: (A syntactical identity.)

```
$ENTRY Go { e.string = <F e.string>; }
F { s.1 = s.1;
    s.1 e.string = <F s.1> <F <F e.string>>;
        = ; }
```

The residual program: `$ENTRY Go { e.input = e.input; }` This example demonstrates that our transformation is able to decrease time complexity from $O(2^n)$ to $O(1)$. The domain of the partial mapping was extended.

Example 2: The following entry point to the function `Repl` defined in Sect. 3

`$ENTRY Go { s.a e.xs = <Repl s.a s.a (e.xs)>; }`, after specialization w.r.t. the call context, will be transformed to the following projection `$ENTRY Go { s.a e.xs = e.xs; }`

9.3. Use of the global analysis results, repeated specialization. If the results of the global analysis allow us to replace a folded component with a one-step program, then its input point is replaced with a corresponding passive node.

Consider the main case. An inductive output format of a recursive folded component `P` was constructed as a result of the analysis: the structures of the output environment (of the considered sub-graph `P`) are specified. We again are at the starting position to specialize the sub-graph

P; because it may contain edge-references to itself and,hence,the information about the output structures was not used during the construction of P.

Repeated specialization of the sub-graph specializes the folded component w.r.t. the structures of the inductive output formats of the sub-graphs F_i, which the component refers to (i.e. there exists an edge coming out of a node of the component and coming in the input point of a F_i).

Example 1: Consider a modification of Example 1 from Sect. 7.2.1. We change only the entry point and the definition of Fact.

```
$ENTRY Go { e.n = <Fact (e.n)>; }
Fact { () = (I);
       (e.n) = (<Times (e.n) <Fact (<Minus (e.n) (I)>)»); }
```

The sub-graph corresponding to this task immediately after the global analysis can be found in Sect. 5.2. (A reference F16 to another component is not relevant and we skip it.) In this example, F7 is specialized w.r.t. the structures of the output formats of F7 and F16. If an output format of a sub-graph F_i has not been constructed yet, then during the specialization the output format is considered in general situation (no information about its structure).

This transformation does not make unfolding: the sub-graph is just cleaned. Even though the specialization deals with the tools of the driving algorithm.

The global analysis performed not only for the folded components but also for every completely folded task. Global information is distributed from one task to another: through the links between them and through the shared parameters.

10. Dead code analysis

Dead code analysis post-processes the "residual" program constructed by the main phase of Scp4. The analysis is a kind of a global dependence analysis that aims to compute the minimum amount of information sufficient for producing certain results. The goal is to eliminate the input and output formal parameters (of the folded components) and the function calls, which values do not influence the values of the partial mapping defined by the Scp4 input.

Example 1: The following program

```
$ENTRY Go { e.1 = <F (e.1) () e.1>; }
F { (e.0) (e.2) A e.1 = <F (<F (e.0) (e.2 C) e.1>) (e.2 B) e.1>;
    (e.0) (e.2)       = e.2; }
```

will be cleaned to

```
$ENTRY Go { e.1 = <F45 (e.1)>; }
F45 { (A e.1) e.2 = <F45 (e.1) e.2 B>;
      ()    e.2 = e.2; }
```

This post-processing is quite critical, because the redundant formal parameters not only are brought by the source program but and may be generated by the algorithm of generalization.

11. Experiments

This section gives several references and short examples of transformations by Scp4. See Appendix A for the examples of specialization of an interpreter w.r.t the examples given in this paper. A lot of examples can be found in the distributive of Scp4 [16].

Example 1: A. V. Korlyukov describes several interesting experiments with Scp4 in [6–10]. For instance, he demonstrates using the supercompiler as a "theorem prover" [8, 15].

Example 2: We refer the reader to [10], where the experiments on supercompilation of a double interpretation are reported. The supercompiler specializes an XSLT-interpreter written in Refal w.r.t. a Turing Machine interpreter written in XSLT. The observed running time speedup is entered in the title of the report. A demonstration of the experiments is available for immediate download [10].

Example 3: Consider `Plus` from Example 1 in Sect. 7.2.1. Specialization of this function w.r.t. the second argument is not a surprise. Let us specialize the function w.r.t. the first argument by the supercompiler. We input the entry point `$ENTRY Go { (e.n) (e.m) = <Plus (I e.n) (e.m)>; }` to Scp4. The residual program looks as follows.

```
$ENTRY Go { (e.n) (e.m) = I <F7 (e.n) e.m>; }
F7 { (e.1)       = e.1;
     (e.1) I e.2 = I <F7 (e.1) e.2>; }
```

Example 4: Let the entry point for `Minus` (Example 1,Sect. 7.2.1) be `$ENTRY Go{(e.n)(e.m) = <Minus (e.n)(I e.m)>;}` The Scp4's output:

```
$ENTRY Go { (I e.n) (e.m) = <F6 (e.n) e.m>; }
F6 { (I e.1) I e.2 = <F6 (e.1) e.2>;
     (e.1)         = e.1; }
```

Example 5: This example demonstrates transformation of a recursive program to the tail recursive variant. The entry point `<Times (e.n) (e.m)>` to `Times` defined in Sect. 7.2.1 (Example 1) is input to Scp4. The residual program:

```
$ENTRY Go { (e.n) (e.m) = <F5 (e.n) e.m>; }
F5 { (e.1) = ;
    (e.1) I e.2 = <F24 (e.1) (e.2) e.1>; }
F24 { (e.5) (e.6) = <F5 (e.5) e.6>;
     (e.5) (e.6) I e.7 = I <F24 (e.5) (e.6) e.7>; }
```

Example 6:
Elimination of intermediate data. The parameterized entry point:

```
$ENTRY Go{ s.a s.b s.c e.inp = <Repl (s.b s.c) <Repl (s.a s.b) e.inp»;}
```

,where `Repl` was defined in Sect. 3. The residual program:

```
$ENTRY Go {s.a s.b s.c e.inp = <F8 (e.inp) s.a s.b s.c>;}
F8 {  () s.1 s.2 s.3 = ;
(s.1 e.4) s.1 s.2 s.3 = s.3 <F8 (e.4) s.1 s.2 s.3>;
(s.2 e.4) s.1 s.2 s.3 = s.3 <F8 (e.4) s.1 s.2 s.3>;
(s.6 e.4) s.1 s.2 s.3 = s.6 <F8 (e.4) s.1 s.2 s.3>;
((e.7) e.4) s.1 s.2 s.3
        = (<F8 (e.7) s.1 s.2 s.3>) <F8 (e.4) s.1 s.2 s.3>; }
```

See also [13] and Appendix A for various examples.

12. Conclusions

We have described the general structure of an experimental program specializer Scp4 based on the supercompilation technique. The input language of the transformer is a functional programming language Refal-5. We have shown the results of a number of examples transformed by Scp4 and gave the references to the reports, where a lot of other examples can be found.

Every concrete tool from Scp4 makes a simplest elementary transformation. A result of the composition of these transformations often is interesting. Some transformations do not depend one on another and can be done in an order, which is different from the one used by Scp4. The transformations do not commutate. That means repeated use of the supercompiler can lead to further optimizations.

Transformation on the basis of the results of the global analysis (and the lazy driving) can decrease time complexity of programs to be transformed.

Non-trivial derivation of the output formats substantially depends on the existence of the references to other components in the local folded-component, such that their output formats are still not created to the given moment. The experiments made by the author show that without

derivation of the output formats interesting transformations do not happen, when the length of the function stack of a program to be transformed is not uniformly bounded on input data.

The termination property of the supercompiler Scp4 depends on the driving strategy chosen by the user. The most conservative strategy causes termination. If another strategy is chosen, then Scp4 may loop forever on some inputs, but will yield more efficient residual programs when it will terminate.

The residual folded-components are often tail recursive. That and some other reasons indicate that, from the practical point of view, the direct interpretation of the Refal-graph language makes meaningful interest (we would like to point to the work by A. P. Konyshev [11]).

References

[1] Burstall R. M., Darlington J. *A transformation system for developing recursive programs.* — vol. **24**, no. 1: ACM Press, 1977, pp. 44–67. ↑

[2] Ershov A. P. *Mixed computation in the class of recursive program schema.* — vol. **4**, no. 1, 1978. ↑1

[3] Futamura Y., Nogi K. *Generalized partial computation* // Of the IFIP TC2 Workshop. — Amsterdam: North-Holland Publishing Co., 1988, pp. 133–151. ↑1

[4] Higman G. *Ordering by divisibility in abstract algebras* 2. — vol. **2**, no. 7, 1952, pp. 326–336. ↑7.2.1

[5] Jones N. D., Gomard C. K., Sestoft P. // Partial Evaluation and Automatic Program Generation: Prentice Hall International, 1993. ↑1

[6] Korlyukov A. V. User manual on the Supercompiler Scp4, 1999, http://www.refal.net/supercom.htm. (Russian) ↑1, 11

[7] Korlyukov A. V. *Automatic deriving of some mathematical formulae* // Of the VIII-th Byelorussian Mathematical Conference: Theses of the lectures, 2000, pp. 178. (Russian) ↑

[8] Korlyukov A. V. A number of examples of the program transformations with the supercompiler Scp4, 2001, http://www.refal.net/~korlukov/pearls/. (Russian) ↑11

[9] Korlyukov A. V. *Supercompilation of the super-spies*: Alpha. — vol. **1**. — Byelorussia: The Grodno State University, 2001, pp. 89–98. (Russian) ↑

[10] Korlyukov A. V., Nemytykh A. P. Supercompilation of Double Interpretation. (How One Hour of the Machine's Time Can Be Turned to One Second), 2002, http://www.refal.net/~korlukov/scp2int/Karliukou_Nemytykh.pdf, Sources, demonstration: http://www.refal.net/~korlukov/demo_scp4xslt.zip. ↑1, 11

[11] Konyshev A. P. The translator from the Refal-graph language to the language C: sources and demonstration, 2000, http://www.botik.ru/pub/local/scp/refal5/. ↑1, 12

[12] Kruskal J. B. *Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture*: Trans. Amer. Math. Society. — vol. **95**, 1960, pp. 210–225. ↑7.2.1

[13] Nemytykh A. P. *Supercompiler scp4: Use of quasi-distributive laws in program transformation* // International Software Engineering Symposium: Wuhan University Journal of Natural Sciences. — vol. **95(1-2)**. — Wuhan, China, 2001, pp. 375–382. ↑1, 5, 9, 9.1, 11

[14] Nemytykh A. P. *A Note on Elimination of Simplest Recursions* // Of the ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation. — Aizu, Japan: ACM Press, 2002, pp. 138–146. ↑1, 9, 9.2

[15] Nemytykh A. P. *Playing on REFAL* // Of the International Workshop on Program Understanding A. P. Ershov Institute of Informatics Systems, Siberian Branch of Russian Academy of Sciences. — Novosibirsk–Altai Mountains, 2003, pp. 29–39, Also available at http://www.botik.ru/pub/local/scp/refal5/korlyukov.html. ↑11

[16] Nemytykh A. P., Turchin V. F. The Supercompiler Scp4: sources, on-line demonstration, 2000, http://www.botik.ru/pub/local/scp/refal5/. ↑1, 7.2.1, 9, 11

[17] Pettorossi A., Proietti M. *Transformation of logic programs: Foundations and techniques*. — vol. **19,20**, 1994, pp. 261–320. ↑1

[18] Romanenko S. A. Refal-4 - an extension of Refal-2, in which the driving can be expressed: Preprint. — vol. **147**. — Moscow: M. V. Keldysh Institute for Applied Mathematics, Russian Academy of Sciences, 1987. (Russian) ↑

[19] Romanenko S. A. *Arity raiser and its use in program specialization* // The ESOP'90. — vol. **432**: Springer-Verlag, 1990, pp. 341–360. ↑1

[20] Sørensen M. H., Glück R. *An algorithm of generalization in positive supercompilation* // Logic Programming: Proceedings of the 1995 International Symposium: MIT Press, 1995, pp. 486–479. ↑7.2.1

[21] Turchin V. F. *The concept of a supercompiler*. — vol. **8**: ACM Press, 1986, pp. 292–325. ↑1, 5, 6.1

[22] Turchin V. F. *The algorithm of generalization in the supercompiler* // of the IFIP TC2 Workshop, Partial Evaluation and Mixed Computation: North-Holland Publishing Co., 1988, pp. 531–549. ↑1, 7, 7.2.1, 1

[23] Turchin V. F. // Refal-5, Programming Guide and Reference Manual. — Holyoke, Massachusetts: New England Publishing Co., 1989,
(electronic version: http://www.botik.ru/pub/local/scp/refal5/ ,2000). ↑1, 3, A

[24] Turchin V. F. Metacomputation in the language Refal, 1990, (unpublished, private communication). ↑1, 1, 6.1

[25] Turchin V. F. *Metacomputation: Metasystem transition plus supercompilation* // Of the PEPM'96. — vol. **1110**: Springer-Verlag, 1996, pp. 481–509. ↑7.2.1

[26] Turchin V. F. *Supercompilation: Techniques and results* // Of Perspectives of System Informatics. — vol. **1181**: Springer-Verlag, 1996, pp. 227–248. ↑1, 1

[27] Turchin V. F., Turchin D. V., Konyshev A. P., Nemytykh A. P. Refal-5: sources, executable modules, 2000, http://www.botik.ru/pub/local/scp/refal5/. ↑1

[28] Wadler P. *Deforestation: Transforming programs to eliminate tree.* — vol. **73**, 1990, pp. 231–238. ↑1

Appendix A. Specialization of an interpreter

The purpose of this section is to give some additional examples of transformations by the supercompiler Scp4.

We are going to specialize an interpreter w.r.t. given programs under the applicative strategy of the unfolding (Sect. 8). So firstly we define an algorithmically full subset of the Refal language (see Sect. 3 and [23]), called strict Refal. The subset will be interpreted. The data is the Refal data. The program syntax is defined by the following grammar.

```
Program ::= $ENTRY definition+
definition ::= function-name { sentence;+ }
sentence ::= pattern = expr
expr ::= empty | term expr1 | function-call expr1
function-call ::= <function-name expr>
pattern ::= empty | term pattern1
term ::= SYMBOL | var | (expr)
var ::= e.name | t.name | s.name
empty ::=                          /* nihil */
```

Syntax of the strict Refal language.

There are two additional restrictions: 1) the set of the variables of the right side of a sentence is a subset of the variable set of the left side of the sentence; 2) two `e`-variables are not allowed on the same parenthesis structure in the patterns. (For example, the pattern `(e.1) e.2 (e.3)` is allowed, while `(e.1 A e.2) e.3` is not.)

The given-below interpreter written in the strict Refal is

$$\texttt{Interpreter: Program} \times \texttt{Data} \mapsto \texttt{Data}_\perp,$$

where `Program` stands for the set of the strict Refal programs, and `Data` stands for the set of the Refal data. We use the following mapping for representing the programs and the data with the Refal data. Denote the mapping by underlining.

```
Program = ( definition+ )
F { sentence;+ } = (F sentence;+ )
pattern = expr;  = (( pattern )'='( expr ))
expr1 expr2 = expr1 expr2
( expr ) = ('*' expr )
<F expr> = (Call F expr )
```

```
e.name = (Var 'e' name)
t.name = (Var 't' name)
s.name = (Var 's' name)
SYMBOL = SYMBOL
```

A pattern is the partial case of an expression. The empty expression is encoded by itself. Below the asterisk on the first position in a line stands for the one-line comments.

```
$ENTRY Go {
t.Program e.data = <Interpreter (Call Go e.data) t.Program>;
}
Interpreter {
(Call s.F e.d) t.P = <Eval <EvalCall s.F (e.d) t.P> t.P>;
}
* <Eval (e.env) (e.expr) t.Program> => e.data
Eval {
(e.env) ((Call s.F e.expr1) e.expr) t.P
   = <Eval <EvalCall s.F (<Eval (e.env) (e.expr1) t.P>) t.P> t.P>
     <Eval (e.env) (e.expr) t.P>;
(e.env) ((Var e.var) e.expr) t.P
      = <Subst (e.env) (Var e.var)> <Eval (e.env) (e.expr) t.P>;
(e.env) (('*' e.expr1) e.expr) t.P
     = ('*' <Eval (e.env) (e.expr1) t.P>) <Eval (e.env) (e.expr) t.P>;
(e.env) (s.x e.expr) t.P = s.x <Eval (e.env) (e.expr) t.P>;
(e.env) () t.P = ;
}
EvalCall { s.F (e.d) t.P =
  <Matching False (((False)'='(False)) <LookFor s.F t.P>) (e.d)>;}
* <Matching t.boolen (e.pattern)'='(e.data)> => (e.env) (e.expr)
* , where t.boolen ::= (e.env) | False
Matching {
False (t.sent ((e.p)'='(e.expr)) e.def) (e.d) =
          <Matching <RigitMatch (e.p)'='(e.d) ()>
                    (((e.p)'='(e.expr)) e.def) (e.d)>;
(e.env) (((e.p)'='(e.expr)) e.def) (e.d) = (e.env) (e.expr);
}
* <RigitMatch (e.patt)'='(e.data) (e.env)> => (e.env1) | False
RigitMatch {
((Var 'e' s.n))'='(e.s) (e.env)
  = <RigitMatch ()'='() <PutVar ((Var 'e' s.n) e.s) (e.env)»;
((Var 's' s.n) e.p)'='(s.1 e.s) (e.env)
  = <RigitMatch (e.p)'='(e.s) <PutVar ((Var 's' s.n) s.1) (e.env)»;
((Var 't' s.n) e.p)'='(t.1 e.s) (e.env)
```

```
  = <RigitMatch (e.p)'='(e.s) <PutVar ((Var 't' s.n) t.1) (e.env)»;
(('*' e.p1) e.p)'='(('*' e.1) e.s) (e.env)
  = <RigitMatch (e.p)'='(e.s) <RigitMatch (e.p1)'='(e.1) (e.env)»;
(s.1 e.p)'='(s.1 e.s) (e.env)
  = <RigitMatch (e.p)'='(e.s) (e.env)>;

((Var 'e' s.n) e.p (Var 's' s.n1))'='(e.s s.1) (e.env)
     = <RigitMatch ((Var 'e' s.n) e.p)'='(e.s)
         <PutVar ((Var 's' s.n1) s.1) (e.env)»;
((Var 'e' s.n) e.p (Var 't' s.n1))'='(e.s t.1) (e.env)
     = <RigitMatch ((Var 'e' s.n) e.p)'='(e.s)
         <PutVar ((Var 't' s.n1) t.1) (e.env)»;
((Var 'e' s.n) e.p ('*' e.p1))'='(e.s ('*' e.1)) (e.env)
     =<RigitMatch (e.p1)'='(e.1)
       <RigitMatch ((Var 'e' s.n) e.p)'='(e.s) (e.env)»;
((Var 'e' s.n) e.p s.1)'='(e.s s.1) (e.env)
     = <RigitMatch ((Var 'e' s.n) e.p)'='(e.s) (e.env)>;

()'='() (e.env) = (e.env);
(e.p)'='(e.s) e.False = False;
}
* <PutVar (e.assignment) (e.env) > => t.boolean
PutVar {
 t.a (e.env) = <CheckPut <PutVar1 t.a (e.env)»; }
PutVar1 {
((Var s.t s.n) e.val) (((Var s.t s.n) e.val1) e.env)
         = ((Var s.t s.n) e.val1) e.env <Eq (e.val) (e.val1)>;
t.assign (t.assign1 e.env)
           = t.assign1 <PutVar1 t.assign (e.env)>;
t.assign () = t.assign True;
}
 CheckPut {
e.env True = (e.env);
e.trash False = False;
}
* <Eq (e.expression1) (e.expression2)> => s.boolean
* , where s.boolean ::= True | False
Eq {
(s.1 e.xpr1) (s.1 e.xpr2) = <Eq (e.xpr1) (e.xpr2)>;
(('*' e.1) e.xpr1) (('*' e.2) e.xpr2)
     = <Eq (e.1 e.xpr1) (e.2 e.xpr2)>;
() () = True;
(e.xpr1) (e.xpr2) = False;
```

```
}
* <LookFor s.Function-name (e.Program)> => e.def
LookFor {
s.F ((s.F e.def) e.P) = e.def;
s.F ((s.F1 e.def) e.P) = <LookFor s.F (e.P)>;
}
* <Subst (e.environment) t.variable> => e.data
Subst {
(((Var s.t s.n) e.val) e.env) (Var s.t s.n) = e.val;
(t.assign e.env) t.var = <Subst (e.env) t.var>;
}
```

All parameterized entry points of the below-given tasks to be transformed do not have parameters describing the restriction to the image of the encoding. So the supercompiler solves more general tasks than it is necessary for the first Futamura projection.

Example 1: We input the following parameterized entry point to Scp4: `<Go Go-Plus e.data>`, where `Go-Plus` is the program

```
$ENTRY Go {(e.1) (e.2) = <Plus (e.1) (e.2)>;}
```

, and `Plus` was defined in Sect. 7.2.1. The residual program is

```
$ENTRY Go { ('*' e.1) ('*' e.2) = e.2 e.1;}
```

The result is perfect. There are no loops. The function domain was extended.

Example 2: The input to Scp4 is `<Go Go-Repl e.data>`, where `Go-Repl` is

```
$ENTRY Go { (s.a e.b) e.inp = <Repl (s.a e.b) e.inp>; },
```

and `Repl` was defined in Sect. 3. The residual program looks as follows.

```
$ENTRY Go { ('*' s.a e.b) e.inp = <F64 (e.inp) (e.b) s.a>; }
* InputFormat:  <F64 (e.4) (e.1) s.3 >
F64 {
() (e.1) s.3 = ;
(s.9 e.4) (e.1) s.9 = e.1 <F64 (e.4) (e.1) s.9>;
(s.9 e.4) (e.1) s.3 = s.9 <F64 (e.4) (e.1) s.3>;
(('*' e.6) e.4) (e.1) s.3
        = ('*' <F64 (e.6) (e.1) s.3>) <F64 (e.4) (e.1) s.3>;
}
```

The interpretation overhead was completely removed.

Example 3: The parameterized entry point is

```
<Go Repl1 e.inp>,
```

where `Repl1` is the function `Go` defined in Sect. 3. The residual program:

```
$ENTRY Go { e.inp = <F42 e.inp>; }

* InputFormat: <F42 e.1>
F42 {
              = ;
 Lisp     e.1 = Refal <F42 e.1>;
 s.2      e.1 = s.2 <F42 e.1>;
('*' e.2) e.1 = ('*' <F42 e.2>) <F42 e.1>; }
```

The interpretation overhead was completely removed. Moreover, the interpreted program was specialized.

Example 4: The input to Scp4 is `<Go Go-Times e.data>`, where `Go-Times` is the program

```
$ENTRY Go {(e.1) (e.2) = <Times (e.1) (e.2)>;},
```

and `Times` was defined in Sect. 7.2.1. The residual program is the following.

```
$ENTRY Go {('*' e.1) ('*' e.2) = <F56 (e.1) e.2>;}

* InputFormat: <F56 (e.1) e.2>
F56 { (e.1)  = ;
      (e.1) I e.2  = e.1 <F56 (e.1) e.2>; }
```

There is no interpretation overhead. Moreover, the residual program does not have a loop corresponding to `Plus` from the original program. Time complexity was decreased.

Example 5: The parameterized entry point is `<Go Fact1 e.n>`, where `Fact1` is the function `Go` defined in Sect. 9.3 (Example 1). The residual program:

```
$ENTRY Go { e.n = ('*' <F30 e.n>); }

* InputFormat: <F533 (e.1) (e.4) e.5>
F533 { (e.1) ()  = ;
       (e.1) () I e.5 = I e.1 <F533 (e.1) () e.5>;
}
* InputFormat: <F245 (e.1) e.4>
F245 { (e.1)  = ;
       (e.1) I e.4 = I I e.1 <F245 (e.1) e. 4>;
}
```

```
* InputFormat: <F122 e.1>
F122 { I e.1 = <F245 (e.1) <F122 e.1>>;
       e.1   = <F533 (e.1) () <F30 e.1>>;
}
* InputFormat: <F30 e.1>
F30 {        = I ;
      I e.1  = <F122 e.1>; }
```

The residual algorithm does not have a loop coming out the function `Plus`. The first sentence of `F122` unwraps the function stack according to the original definition of `Fact`. The exit from the recursion looks strange and can be improved. We input the result program to Scp4. The repeated supercompilation just renames these functions and cleans the function `F533` a little:

```
* InputFormat: <F33 (e.1) e.2>
F33 { (e.1)       = ;
      (e.1) I e.2 = I e.1 <F33 (e.1) e.2>; }
```

Thus, the exit from the recursion is still strange.

Example 6: We input the following parameterized entry point to Scp4: `<Go F1 e.1>`, where `F1` is the function `Go` defined in Sect. 10 (Example 1). The residual program is

```
$ENTRY Go { e.1 = <F100 () e.1>; }


* InputFormat: <F100 (e.8) e.9>
F100 { (e.8) A e.9 = <F100 (e.8 B) e.9>;
       (e.8)       = e.8; }
```

Comments are unnecessary.

The following table shows the times taken by the supercompiler Scp4 for producing the residual programs given in this section. All the experiments were performed under the operating system *Microsoft Windows 2000* with *Intel Pentium-4 CPU (R), 262 144 KB RAM, 1.5 GHz.*

The supercompile times.

Example	1	2	3	4	5	6
Time in sec.	0.81	3.28	2.73	1.69	15.89	1.67

Исследовательский центр мультипроцессорных систем ИПС РАН и Уханьский университет, г. Ухань, Хубей, Китай

А. П. Немытых. *Суперкомпилятор SCP4: общая структура.* (Англ.)

Аннотация. На основе технологии суреркомпиляции автор реализовал преобразователь фунуциональных программ Scp4. Scp4 реализован на функциональном языке программирования Рефал-5. Этот же язык является и входным языком для Scp4. В статье мы кратко рассматриваем общую структуру суперкомпилятора Scp4 и показываем несколько примеров преобразований посредством Scp4. Библ. 28 наим.