Program Transformation with Metasystem Transitions: Experiments with a Supercompiler

Andrei P. Nemytykh and Victoria A. Pinchuk

Program Systems Institute, Pereslavl-Zalesski, Yaroslavl Region, Russia, 152140

E-mail: nemytykh{vika}@scp.botik.yaroslavl.su, Fax 08535-20593

Abstract. *Turchin's supercompilation* is a program transformation technique for functional languages. A *supercompiler* is a program which can perform a deep transformation of programs using a principle which is similar to *partial evaluation*.
In the present paper we use a supercompiler, which V.F. Turchin and we have described in [22], [23]. The aim of our investigation has been to show, what deep changes (w.r.t. run time) in the programs can be achieved by supercompilation.
In [21] V.F. Turchin presented a method to improve the transformational power of supercompilation without modifying the transformation system. We use this idea to show both the power of the method and abilities of our supercompiler. Our examples include a generation of both one-step unfolding and instantiations, lazy evaluation, inverse evaluation.

Keywords: program transformation, optimization, recursion, supercompilation, metacomputation, metasystem transition, Refal.

1 Introduction

Turchin's supercompilation is a program transformation technique for functional languages. A *supercompiler* is a program which can perform a deep transformation of programs using a principle which is similar to *partial evaluation*.

In the present paper we use a supercompiler, which V.F. Turchin and we have described in [22], [23]. The supercompiler (`Scp`) is the result of the ground breaking investigations by V.F. Turchin [17], [18], [19], [20]. The present version of `Scp` differs from the one presented before [22], [23] in one point. This point is a *whistle and generalization*. *Whistle* is a method to stop the unfolding looking for a possible cycle. *Generalization* is a method to find the most specific generalization of terms seen in the unfolding with the purpose to manage the cycle. In contrast with [22], [23], our *whistle* has no a termination property for all programs to be transformed.

Many researchers emphasize the termination aspect of the program transformations. In our opinion it is a nonsense to give any termination property, while we have no statements about the residual programs. Here we consider the program transformations w.r.t. time of computation. The aim of our investigation has been to show, what deep changes (w.r.t. run time) of programs can be achieved by supercompilation when it terminates. We do not present here a general method of showing how our transformations improve efficiency. However, we show that our examples are clean in some natural sense.

In [21] V.F. Turchin presented a method to improve the transformational power of supercompilation without modifying the transformation system. The main idea is to insert an interpreter between the object program and the transformation system. R. Glück and J. Jørgensen used this idea [8] for generating optimizing specializers. We use this idea to show both the power of the method and the abilities of our supercompiler. Our examples include a generation of the one-step unfolding and instantiations, lazy evaluation, inverse evaluation.

2 Preliminaries

2.1 Object language

Our setting is a fragment of Refal. We mean this fragment, whenever we say Refal.

Refal:
Refal is a first-order functional language with an applicative order (inside-out) semantics [17], [20].

Definition:
(1) *Data* of Refal is the set of all finite sequences of *object terms*. The empty sequence belongs to *Data*.
(2) An *object term* is a *(sequence of object terms)* or *symbol* .
(3) A *symbol* is a *number* or *' character '* or *symbolic-name*, which is not a *variable*.
(4) An *object expression* is a *datum*. □

Every program is a term rewriting system. The semantics of Refal is based on pattern-matching. As usual, the rewriting rules are ordered to match from top to bottom. There are two types of variables in Refal: *s-variables*, such as `s.1` or `s.x`, take exactly one symbol as its value; *e-variables*, as `e.2`, can have any object expression as its value. A variable can not be on the right hand side, if it is absent on the left hand side of a rule. The arity of any function equals one.

Example 1:

```
Sub { (e.Z) (e.X) = <Sub1 (e.X) () (e.Z)>; }
Sub1 { (e.Y) (e.Y) (e.Z)     = (e.Z);
       (e.Y) (e.C) (e.Z '1') = <Sub1 (e.Y) ('1' e.C) (e.Z)>; }
```

This program defines the function of the unary subtraction: $(e.Z) - (e.X)$. A nutural number is a sequence of $'1'$ enclosed in parentheses.

Strict Refal:
Definition:

A *rigid pattern* is a pattern such that (a) none of its subpatterns has the form $E_1\mathtt{e}.i_1 E_2\mathtt{e}.i_2 E_3$, where E_1 etc. are arbitrary patterns (we say that there are no *open* variables like $e.i_1$ and $e.i_2$ here), and (b) no e-variable appears in the pattern twice (we say that there are no *repeated* variables). □

Strict Refal is Refal where only rigid patterns are allowed. The program of the *example 1* is written in non-strict Refal. The following one is in the strict.

Example 2:

```
Fabc { e.X = <Repl ('b' 'c') <Repl ('a' 'b') e.X>>; }
Repl { (s.1 s.2) s.1 e.X = s.2 <Repl (s.1 s.2) e.X>;
       (s.1 s.2) s.3 e.X = s.3 <Repl (s.1 s.2) e.X>;
       (s.1 s.2)   = ; };
```

Repl is a function to replace any symbol $s.1$ by $s.2$ in a string $e.X$.
$< Fabc$ 'a' 'd' 'b' $>=$ 'c' 'd' 'c'.

Flat Refal:
Flat Refal is a fragment of the strict Refal. The right-hand side of every Refal rule in *flat Refal* is either a pattern or a single function call; nested function calls are not allowed. The programs of the *examples 1 and 2* are not in flat Refal. The next one is in the flat. Flat Refal is computationally complete.

Example 3:

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

Our supercompiler uses *flat Refal* as the language of object programs.

2.2 Task language

As a task language to transform of the programs we use *MST-schemes* [22], [23]. For instance, the following MST-scheme:

```
<Scp ................>
    <Int ... e.data  >
       <Prog        >
```

means that `Scp` has to transform the interpreter `Int` which is applied to the given function `Prog`. The program is implicit on the third level of the staircase, the interpreter is implicit to be on the second level. The first call to run by `Int` is the function call of `Prog` , the first call to run the interpreter itself is $< Int\ Prog, e.data >$. `e.data` is a parameter to describe a class of tasks: we have no information about data of `Prog` . Hence, nothing is known about the second argument of `Int` . The result of `Scp` will be some program, which represents a function on `e.data` .

3 Program Transformation

Definition:
(1) A *datum-configuration* with respect to a program P is a *object expression* or *datum-config.*$_1$ $<$ *Symbolic-name datum-config.*$_2$ $>$ *datum-config.*$_3$. The *Symbolic-name* is a name of a function defined in the program.
(2) DC_P is the set of the data-configurations w.r.t. P.
(3) $Conf_P$ is some datum-configuration w.r.t. P. □

Definition:
One-step unfolding w.r.t. a program P is a function:

$$UnF : DC_P \cup \{Z\} \mapsto DC_P \cup \{Z\}.$$

Let $<$ *Func object expr.* $>$ be the current function call to evaluate (inside-out, left first), let $L = R$ be a rewriting rule of the $Func$ successfuly matched with the object expr. , then

$$UnF[x] = \begin{cases} x, \text{ if } x \text{ is an object expression;} \\ Z, \text{ if } x \text{ is } Z; \\ \text{the result of replacement of the occurence of the current} \\ \text{function call with } R \text{, where the substitution} \\ \text{for variables from the matching was done;} \\ Z, \text{ otherwise.} \end{cases}$$

□

Definition:
The evaluation *time* ($Time_P^{Conf}$) of $Conf_P$ is

$$min\{n \in N \mid \underbrace{\boldsymbol{UnF}[\boldsymbol{...UnF}}_{n}[Conf_P\ \underbrace{]...]}_{n} = x, where\ x\ is\ a\ datum\ or\ Z\}\ ,$$

if the minimum exists, and ∞ otherwise. □

Let $\mathcal{CONF_P} ::= Conf_P^{\Lambda}$ be a parametrized set of the configurations w.r.t. some program P, where Λ is some set of parameters. Then it defines a function $Func_{\mathcal{CONF_P}} : \Delta \mapsto Data \cup \{Z\}$ in a natural way; $Func_{\mathcal{CONF_P}}[\lambda_0]$ is the result of evaluation of the configuration $Conf_P^{\lambda_0}$. Here Δ is the subset of the domain of Λ, where the computation is not looping forever.

Definition:
We say that $(Prog_1, Conf^{\Lambda}_{Prog_1}) \succeq (Prog_2, Conf^{\Lambda}_{Prog_2})$, iff (1) $\Delta_1 \subseteq \Delta_2$, (2) the corresponding functions coincide on Δ_1, (3) $Time^{Conf^{\lambda}}_{Prog_1} \geq Time^{Conf^{\lambda}}_{Prog_2}$ for all $\lambda \in \Delta_1$. We say that $(Prog_1, Conf^{\Lambda}_{Prog_1}) \succ (Prog_2, Conf^{\Lambda}_{Prog_2})$, iff, in addition, there exists $\lambda_0 \in \Delta_1$ such that $Time^{Conf^{\lambda_0}}_{Prog_1} > Time^{Conf^{\lambda_0}}_{Prog_2}$. □

Here we consider program transformations relatively to the evaluation time. Many researchers emphasize the termination aspect of the program transformations. In our opinion, it is a nonsense to give any termination property, while we have no statements about the residual programs. So our system does not pretend to have the termination property for all programs to be transformed. The aim of our investigation is to show what deep changes (w.r.t. run time) in the programs can be performed by supercompilation when it terminates.

As far as we know there are only Wadler's papers discuss the properties of the transformed programs [24]. We do not present here a general method of showing how our transformations improve efficiency. We consider this question in [10], [11], [12]. However, we show here that our examples are the clean ones in some natural sense.

3.1 The supercompiler

Our working tool is the supercompiler, which V.F. Turchin and we have described in [22], [23]. We refer to it `Scp`. `Scp` is the result of the ground breaking investigations by V.F. Turchin [17], [18], [19], [20].

The present version of `Scp` differs from the one presented before [22], [23] in one point. This point is a *whistle and generalization*. We consider them in [10], [11], [12]. In contrast with [22], [23] the *whistle* has no a termination property.

We want to stress that our tests were run in a completely automatic fashion; no adjustments have been done. All the tests were run with the same whistle and generalization.

To show how large are our interpreters to be transformed, we give their Refal-sources in [13].

4 Examples of Use of the Metasystem Transitions

A *metasystem* is a program that examines, controls and manipulates other program as an object. A *metasystem transition* is a transition from a program S to a metasystem S' which incorporates the original program as object [17], [20], [21], [7]. In [21] V.F. Turchin presented a method to improve the transformational power of supercompilation without modifying the transformation system. The main idea is to insert an interpreter between the object program and the transformation system. By hand, he gave a number of exapmles to demonstrate it. R. Glück and J. Jørgensen used this idea [8] for the generating optimizing specializers.

We use this idea to show both the power of the method and the abilities of our supercompiler.

4.1 Unfolding

In the first place we consider a very simple transformation: a generation of an additional unfolding.

Let us denote our task to transform as:

```
<Scp ................>
     <Unf-Int . e.x  >
            <Fab   >
```

See Sec.2.1 for the definition of `Fab` . Here e.x is a parameter. `Unf-Int` is some interpreter of flat Refal (Sec.2.1) [13]. The interpreter is only the semantic function of Refal but it is the specific one if we mean a process of computation. We have the result of the specialization of `Unf-Int` with respect to the Fab-function:

```
F2ab { (e.1) = <F1C1 (e.1)()> ; }

F1C1 { (e.22 )('a' 'a'e.1 ) = <F1C1 (e.22 'b' 'b')(e.1 )> ;
       (e.22 )('a') = e.22 'b';
       (e.22 )() = e.22 ; }
```

• We see that $(Unf - Int, Conf^{e.x}_{Unf-Int}) \succ (F2ab, Conf^{e.x}_{F2ab})$, but *we have a much better result:* $(Fab, Conf^{e.x}_{Fab}) \succ (F2ab, Conf^{e.x}_{F2ab})$. There are no redundant pieces from the interpreter in the resulting program.

4.2 Instantiation

It has been known for a long time that specialization of a "naive" algorithm to search for a matching pattern within a string gives the KMP algorithm [9] if the pattern is fixed up. In [6] a pattern-matcher generator was considered, when all what one needs to know about the pattern is its length. This generator was generated in a semi-automatic manner, the source of this KMP-generator was not presented.

We have automatically got the KMP algorithm from the "naive" algorithm, if we know the length of the pattern only. We have got an automatic classification of the set of all patterns (of a given length) with a purpose to do further optimizations for each class independently of others. In the example below the length is two. In this case the classification consists of two classes: when two symbols of the pattern are equal and when they are not. For the first class it is possible to do some optimization of the "naive" algorithm while for the second class the optimization is impossible.
The program to represent the "naive" algorithm was written as:
Example 4:

```
Search { (s.X s.Y)  s.X s.Y e.S = TRUE;
         (s.X s.Y)  s.Z e.S = <Search (s.X s.Y) e.S>;
         (s.X s.Y)   = FALSE; }
```

The MST-scheme for our task is:

```
<Scp ................................. >
     <Class-Int ....... s.1 s.2  e.x  >
              <Search (  ↓    ↓ )   ↓ >
```

Here s.1, s.2, e.x are parameters. s.1 and s.2 describe a pattern and e.x describes a string. Hence, the pattern length equals two. `Class-Int` is a interpreter of flat Refal (Sec.2.1) [13]. The interpreter is the semantic function of Refal only but it is the specific one if we mean a process of computation.

In this case we have the result of the specialization of `Class-Int` with respect to the Search-function:
Input: <KMP s.1 s.2 (e.x)>

```
KMP {
s.1 s.1 (e.3) = <F1C1 s.1 (e.3)>;
s.1 s.2 (e.3) = <F1C3 s.1 s.2 (e.3)>;
}
```

```
F1C1 {
s.1 (s.1 s.1 e.3) = TRUE;
s.1 (s.1 s.4 e.3) = <F1C1 s.1 (e.3)>;
s.1 (s.1)         = FALSE;
s.1 (s.4 e.3)     = <F1C1 s.1 (e.3)>;
s.1 ()            = FALSE; }
```

```
F1C3 {
s.1 s.2 (s.1 e.3) = <F1C2 s.1 s.2 (e.3)>;
s.1 s.2 (s.2 e.3) = <F1C3 s.1 s.2 (e.3)>;
s.1 s.2 (s.4 e.3) = <F1C3 s.1 s.2 (e.3)>;
s.1 s.2 ()        = FALSE; }
```

```
F1C2 {
s.1 s.2 (s.2 e.3) = TRUE;
s.1 s.2 (s.1 e.3) = <F1C2 s.1 s.2 (e.3)>;
s.1 s.2 (s.4 e.3) = <F1C3 s.1 s.2 (e.3)>;
s.1 s.2 ()        = FALSE; }
```

• We see that $(Class-Int, Conf^{s.1,s.2,e.x}_{Class-Int}) \succ (KMP, Conf^{s.1,s.2,e.x}_{KMP})$, but *we have a much better result:* $(Search, Conf^{s.1,s.2,e.x}_{Search}) \succ (KMP, Conf^{s.1,s.2,e.x}_{KMP})$
There are no redundant pieces from the interpreter in the program.

4.3 Lazy evaluation

We have a lazy interpreter (outside-in) of strict Refal (Sec.2.1), [13] to be able to work with the nested calls by means of a MST-scheme:

```
<Scp ........................................>
    <Lazy-Int . s.3  s.4 ........ s.1  s.2 e.x  >
        <Repl (           ) <Repl (          )   >>
```

Function `Repl` was defined in Sec. 2.1. Here the lazy interpreter has to calculate the composition of `Repl` with itself. This composition changes a symbol s.1 into s.2 and after that s.3 into s.4 in a string e.x .
The resulting program is:

```
Fs1s2s3s4 { s.1 s.2 s.3 s.4 (e.1) = <F1C1 () s.1 s.2 (e.1) s.3 s.4>;}
F1C1 {
(e.25) s.1 s.3 (s.1 e.26) s.3 s.4 = <F1C1 (e.25 s.4) s.1 s.3 (e.26) s.3 s.4>;
(e.25) s.1 s.2 (s.1 e.26) s.3 s.4 = <F1C1 (e.25 s.2) s.1 s.2 (e.26) s.3 s.4>;
(e.25) s.1 s.2 (s.3 e.26) s.3 s.4 = <F1C1 (e.25 s.4) s.1 s.2 (e.26) s.3 s.4>;
(e.25) s.1 s.2 (s.29 e.26) s.3 s.4 = <F1C1 (e.25 s.29) s.1 s.2 (e.26) s.3 s.4>;
(e.25) s.1 s.2 () s.3 s.4 = e.25 ; }
```

• Again, we see that
$(Lazy-Int, Conf^{s.1,s.2,s.3,s.4,e.x}_{Lazy-Int}) \succ (Fs1s2s3s4, Conf^{s.1,s.2,s.3,s.4,e.x}_{Fs1s2s3s4})$, but *we have a much better result: the lazy operational semantics was projected into the resulting program.* We have obtained an one-pass program. There are no redundant pieces from the interpreter in the program.

4.4 Inverse evaluation

Two types of the recursive structures of programs:
From *the point of view of the formal Refal syntax*, there are two kinds of recursive definitions w.r.t. an e-variable. These kinds correspond to termination conditions of recursion. The first condition is *a uniform restricted* w.r.t. data, so it cannot be expressed in terms of e-variables, but only in terms of constants (data) and s-variables. The treatment of the second kind of conditions depends on the size of a datum that will be calculated to the moment of the testing. Here we restrict ourselves by simple programs. (See [12] for the details.)

Consequently, the inductive step of the first recursive definition exhausts (decreases) of a value of the e-variable (let it be e.x). The base of the induction is to check some condition *on constants and s-variable's values.* We can treat such a recursive evaluation only as the evaluation from up to bottom. As a rule, $Conf^{A}_{Prog}$ to calculate depends on *some e-parametrs*, which describe domain of e.x. We refer to the first kind of the recursive definitions as to *recursion with exit by exhaustion.* The next example demonstrates this kind of a definition.
Example 5:

```
Input { (e.x)  (e.y)  = <Sum  (e.x) (e.y) >; }

Sum { ('1' e.x) (e.y) = <Sum (e.x) (e.y '1')>;
      ()        (e.y) =  (e.y) ; };
```

This program defines the function of the unary addition: $(e.X) + (e.Y)$. The second sentence of `Sum` shows an exit by exhaustion from the recursion w.r.t e.x. Only thing, which we have to verify, is to test of e.x by matching it with a constant. Here this constant is *empty sequence* .

The inductive step of the second kind of a recursive definition accumulates (increases) of a value of an e-variable. The base of the induction is to check some condition on values of e-variables. There is an hidden recursion in the corresponding pattern. We can treat such a recursive evaluation only as the evaluation from bottom to up. As a rule, $Conf^{A}_{Prog}$ to calculate depends on *constants and s-parametrs* only. We refer to the second kind of the recursive definitions as *recursion with exit by accumulation.* The *example 1* (Sec. 2.1) gives us this kind of a definition. The first sentence of `Sub1` shows an exit by accumulation from the recursion w.r.t `e.C` . We are in need to test that `e.C` equals `e.Y` . It is impossible either in the flat or in the strict Refal. So in this case we have to emulate the second type of recursion by the first one.

An inverse task w.r.t. a given one:
Let Prog be a program. Let $Conf^{A}_{Prog}$ be a parametrized set of configurations and $d \in Data$. Let us consider the problem of resolution of the equation $Conf^{A}_{Prog} = d$. Here A are the unknowns. There are different methods to approach this problem [1], [15], [21]. We will deal with a method, which is known as *inversion.* The main idea is to execute the function definition from its ends to its beginnings, and to compute the value of the input when an output is given (see [15], [21], [12] for the details).

Constraints on a language to be interpreted:
Below we deal with a fragment of flat Refal (see Sec. 2.1).

(1) The right side of each sentence, or the argument of a function call in the right side, must not have of its subpattern has the form $E_1\texttt{e}.i_1E_2\texttt{e}.i_2E_3$, where E_1 etc. are arbitrary patterns. (2) All variables of the left side of a sentence must be met also in the right side.

The tests:
We have an inverse interpreter (`Inv-Int`), which has to resolve the problem (see [12], [13]). In our tests we use a version of `Inv-Int` , which stops after finding the first solution.

The first test, that we want to show, is:

Test 1:

```
<Scp .......................... >
    <Inv-Int ........ (e.out) >
            <Fab X >
```

Function `Fab` was defined in Sec. 2.1. The function has both to replace `'a'` by `'b'` in a given string and to verify, whether this string contains the `'a'` symbols only. Here `e.out` is a parameter to describe the right-hand side of the equation $< Fab\ X >= d$; X is the unknown string.

The program was transformed into the following one:

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

F1C1 { (e.1 'b')(e.25 ) = <F1C1 (e.1)('a'e.25)> ;
       ()(e.25 ) = e.25 ; }
```

• We see that $(Inv-Int, Conf^{e.out}_{Inv-Int}) \succ (Fba, Conf^{e.out}_{Fba})$, but *we have a much better result: the inverse operational semantics was projected into the resulting program.* There are no redundant pieces from the interpreter in the program.

Test 2:

```
<Scp .............................. >
    <Inv-Int ..... e.x .... (e.out) >
          <Input (   ) (Y)>
```

The function `Input` was defined in the *example 5*. Here `Inv-Int` resolves the equation: $< Input\ (d_2)\ (Y) >= (d_1)$, where (d_1) and (d_2) are some given unary natural numbers, Y is the unknown unary natural number. By definition, $(d_2) + (Y) = (d_1)$, so we have $(Y) = (d_1) - (d_2)$ as the output of the interpreter.

As the result of supercompilation, we have the program:

```
Sub {  (e.1) (e.2)= <F1C2 (e.2)()(e.1)>; }
F1C2 {
(e.2)(e.37)(e.38 '1') = <F1C1 (e.37)(e.38)(e.2)(e.2)(e.37)>;
(e.2)(e.37)(e.38) = <F1C3 (e.2)(e.37)(e.38)(e.2)(e.37)>; }
F1C1 {
(e.37)(e.38)(e.2)(s.75 e.72)(s.75 e.73) = <F1C1 (e.37)(e.38)(e.2)(e.72)(e.73)>;
(e.37)(e.38)(e.2)()() = e.38 '1';
(e.37)(e.38)(e.2)(e.72)(e.73) = <F1C2 (e.2)('1' e.37)(e.38)>; }
F1C3 {
(e.2)(e.37)(e.38)(s.263 e.185)(s.263 e.186) = <F1C3 (e.2)(e.37)(e.38)(e.185)(e.186)>;
(e.2)(e.37)(e.38)()() = e.38 ; }
```

So we have obtained a program for unary subtraction $(e.1) - (e.2)$. To see the resulting algorithm, let us rewrite the program in a certain higher level language, named Refal. We obtain the same program as in *the example 1* (Sec. 2.1). Functions `F1C1` and `F1C3` compare two e-variables (see the first sentence of `Sub1` in *the example 1*).

• There is an interesting question here: why *the algorithm* had come from the program transformation. Recalling that (1) the language to interpret by `Inv-Int` is the indicated fragment of *strict* Refal, (2) the target language of `Scp` is flat Refal (i.e. again some fragment of *strict* Refal). So there exist neither *open* variables nor *repeated* ones (see Sec. 2.1). As a consequence, there exists no *syntactic* recursion with exit by accumulation. This, in its turn, forces any entrance in the recursion to be only by accumulation. (For instance, see the definition of `Sum` and `e.x` in the *example* 5, Sec. 4.4.) By definition, the inversion exchanges outputs with inputs. Hence it exchanges the entrances of the recursions by their exits. Hence, *by semantics* , an algorithm of any inversed program can contain the recursions with exit by accumulation only. But our target language does not allow such kind of recursions in its *syntax*. We are forced to encode the exits-by-accumulation in the form of some program. In our test, the exits were encoded in `F1C1` and `F1C3` .

• Conclusion: by definition of inversion, we cannot expect a more efficient algorithm, than the one we have obtained above. Again we see that $(Inv\!-\!Int, Conf^{e.x,e.out}_{Inv-Int}) \succ (Sub, Conf^{e.x,e.out}_{Sub})$. The inverse operational semantics was projected into the resulting program. There are no redundant pieces from the interpreter in the residual program.

5 Conclusion

What we have shown in these tests is both the abilities of our system for program transformations and the power of the Turchin's method of Metasystem Transitions. We gave examples of non-trivial transformations of big programs (non-standard interpreters). The supercompiler produces small residue programs from the big ones. The residue programs have a run time much less than the initial ones. We have shown that with this method such transformations become possible which a direct application of the supercompiler cannot perform. In particular, *we have generated the KMP-algorithm from a "naive" algorithm, when we known a length of a pattern only.*

Acknowlegements. This work could not have been carried out without support, very much attention of Prof. V. F. Turchin. We are very grateful to S.V. Chmutov for his interest to our work. It was impossible to do this without him. Our tests were discussed in Pereslavl (Russia). The authors thank the participants – especially Sergei Abramov, Robert Glück, Andrei Klimov, Igor Nesterov, Sergei Romanenko, Morten Sørensen – for useful comments.

References

1. Abramov S.M., Metavychisleniya i logicheskoe programmirovanie (Metacomputation and logic programming). *Programmirovanie*, 3:31-44, 1991. (in Russian).
2. Burstall R. M., J. Darlington. A Transformation System for Developing Recursive Programs. In *Journal of the ACM.* Vol.24,No.1,pp.44-67,1977.
3. Ershov, A.P. On the essence of compilation, In: E.J.Neuhold (Ed.) *Formal Description of Programming Concepts*, pp.391-420, North-Holland, 1978.
4. Futamura, Y., Partial evaluation of computation process – an approach to compiler compiler. *Systems, Computers, Controls*, *2*,5 (1971) pp.45-50.
5. Jones N., Sestoft P., Sondergaard H., An experiment in partial evaluation: the generation of a compiler generator. In: Jouannaud J.-P. (Ed.) *Rewriting Techniques and Applications*, Dijon, France, LNCS 202, Springer, 1985.
6. Glück R. and Turchin V., Experiments with a Self-applicable Supercompiler, CCNY Technical Report, 1989.
7. Glück R. , Towards multiple self-application. In: *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp.309-320, New Haven, Connecticut, ACM Press, 1991.
8. Glück R. , Jørgensen J., Generating optimizing specializers. In: *IEEE International Conference on Computer Languages*, Toulose, France, IEEE Computer Society Press, 1994.
9. Knuth D.E., Morris J.H., Pratt V.R., Fast Pattern Matching in Strings. *In SIAM Journal of Computer* , 6(2) pp. 323-350, 1977
10. Nemytykh A., Fast Pattern Matching in Strings by Program Transformation. *in preparation*
11. Nemytykh A., Pinchuk V., Transformation Programs to Decrease Run Time. *in preparation*
12. Nemytykh A., Pinchuk V., Inversing of functional programs by metasystem transitions. *in preparation*
13. Nemytykh A., Pinchuk V., Interpretive layers in metasystem transitions. *In: ftp.botik.ru (name: anonymous, pwd: YourSecondName), path: pub/local/scp* , 1995.
14. Pettorossi A. and Proietti M., Transformation of logic programs: Foundations and Techniques, *Istituto di analisi dei sistemi ed informatica, R. 369, Novembre 1993*, Italy.

15. Romanenko A., Inversion and Metacomputation. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Yale University, 1991, USA pp. 12-22.
16. Turchin V. F., Klimov A. V. et al , *Bazisnyi Refal i yego realizatsiya na vychislitel'nykh mashinakh.* (Basic Refal and its implementation on computers) GOSSTROY SSSR, TsNIPIASS, Moscow, 1977 (in Russian).
17. Turchin, V.F. *The Language Refal, the Theory of Compilation and Metasystem Analysis*, Courant Computer Science Report #20, New York University, 1980.
18. Turchin, V.F. The concept of a supercompiler, *ACM Transactions on Programming Languages and Systems*, *8*, pp.292-325, 1986.
19. Turchin, V.F. The Algorithm of Generalization in the Supercopmiler, In *ACM Partial Evaluation and Mixed Computation.* Eds. A.P. Ershov, D. Bjorner, N.D. Jones North-Holland, 1988.
20. Turchin V., *Refal-5, Programming Guide and Reference Manual*, New England Publishing Co., 1989.
21. Turchin V.F., Program Transformation with Metasystem Transitions, *J. of Functional Programming*, 3(3) 283-313, 1993.
22. Turchin V., Nemytykh, A. Metavariables: Their implementation and use in Program Transformation, *Technical Report CSc. TR 95-012, City College of the City University of New York*, 1995.
23. Turchin V., Nemytykh A., Pinchuk V., A Self-Applicable Supercompiler, *to appear in Proceedings of PEPM, Dagstuhl, 1996* .
24. Wadler P. Deforestation Programs to Eliminate Trees. *TCS.*, 73: 231-248, 1990.