On Generalization of Lists and Strings in Supercompilation *

Valentin F Turchin
The City College of New York

1 Introduction

Supercompilation [25, 26, 7, 20, 8, 21, 22, 17, 29, 30] is a technique of equivalent transformation of programs based on *driving* [23, 24, 12], which can be seen as forceful unfolding of function calls, even in cases where it is impossible to choose between several alternatives. Supercompilation is a close relative of several other transformation techniques: *partial evaluation* [4, 3, 9, 10, 11, 5], *deforestation*,[31], and similar techniques based on logic programming, [15, 13, 6, 16] and, especially, [14, 19]

The basic principle of supercompilation is: drive the initial configuration of a computing machine (a function call, possibly nested, which may include free variables), and make the resulting graph of configurations and transitions finite by reducing some configurations to their predecessors, after generalizing some of the predecessors if necessary. Correct generalization is a key to obtaining an optimized program. It is one of the central problems of supercompilation.

Generalization breaks down, naturally, in two parts:

1. Decision to generalize or not generalize the current configuration C with one of its predecessors C' with a view of reducing C to the generalized configuration C_{gen}; this predicate of C and C' is known as a *whistle* which warns us that if we do not try reduction, we may end up with infinite driving.

2. Generalization proper, i.e. the choice of a configuration C_{gen} such that both C and C' are its subsets.

*This material is based upon work supported by the US Civilian Research and Development Foundation under Award No RM1-254

In [27] an algorithm of whistling was defined for lazy evaluation of nested function calls, and termination of supercompilation with this whistle was proven. This algorithm was implemented in a supercompiler and worked very well within its domain of applicability, which is generalization of stacks which represent nested function calls (see, e.g., [30]).

With regard to generalization of *flat* configurations, i.e. ones without nested function calls, the algorithm of [27] used a very simple strategy: if a variable has the same value in both configurations, then it has the same value in the generalization. Otherwise, the variable in the generalization remains free. This strategy guarantees termination (the whistle is given for any pair of configurations with the same function, and the number of functions is finite), but for problems where a non-trivial structure of arguments is essential it leads to over-generalization, hence the supercompilation would not do the transformation that could be expected. Thus, the problem of generalization of flat configurations, or just data structures, remains. Since in the general case (and most advanced supercompilers) we have to intelligently generalize both stacks of function calls, and the constituing function calls themselves, this problem is complementary to the one considered in [27].

Sørensen and Glück [22] use the Higman-Kruskal theorem on homeomorphic embedding, HK for short, (see [2]) to define a whistle for supercompilation which is of proven termination. Their data domain is the set of functional terms with fixed arity of each functional symbol. This kind of data is sufficient for the languages, such as Lisp and Prolog, which operate on binary trees referred to as lists. But the data structures of Refal, which is used as the basic programming language in several existing supercompilers, does not belong to this category. We refer to this structures as *R-expressions* and *R-terms.* An R-expression is a concatenation of any number of R-terms. An R-term is either an elementary syntax unit, or an R-expression parenthesized. This can be seen as the use of a functional symbol of arbitrary arity (a *varyadic* symbol).

Fortunately, HK allows varyadic symbols, so the first thing we do in this paper is define a whistle for supercompilation in Refal following the line of [22]. Then we define and compare the algorithms for evaluating homeomorphic embedding on R-expressions and lists. We show that although in a written form lists and R-expressions are similar, the algorithms for them are different, and even are of different time complexity. After that we consider and compare generalization proper on these two kinds of data. We find considerable differences between them and show that these differences are consequential for the performance of a supercompiler. Finally, we prove that under certain natural assumptions the supercompilation process using the defined algorithms is bound to terminate.

2 Homeomorphic embedding

In this section we follow the work by Dershowitz [2].

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

Definition 1 The *homeomorphic embedding relation* $\trianglelefteq$ on a set $T(F)$ of terms is defined as follows:
Base:

$$s \trianglelefteq s \quad \text{for all } s \in F$$

Recursion. The relation:

$$t = g(t_1, t_2, \ldots, t_n) \trianglelefteq f(t'_1, t'_2, \ldots, t'_m) = t'$$

holds if either:

$$t \trianglelefteq t'_i \quad \text{for some } i = 1, \ldots, m$$

or:

$$f = g \quad \text{and} \quad t_j \trianglelefteq t'_{i_j} \quad \text{for all } j = 1, \quad , n$$

where $1 \leq i_1 < i_2 < \quad < i_n \leq m$. □

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

Theorem 1 (Higman,Kruskal) *If F is a finite set of function symbols, then any infinite sequence $t_1, t_2,$ of terms in the set $T(F)$ of terms over F contains two terms t_j and t_k, where $j < k$, such that $t_j \trianglelefteq t_k$.* □

The definition of the set $T(F)$ and the relation $\trianglelefteq$ in HK can be extended by including in F as constants all natural numbers $n \in N$, and adding to the base of the definition of $\trianglelefteq$ the clause:

$$n_1 \trianglelefteq n_2 \quad \Leftrightarrow \quad n_1 \leq n_2 \qquad \text{for all } n_1, n_2 \in N$$

Theorem 2 *Theorem 1 holds for the extended relation of embedding.*

Proof Define the unary representation of natural numbers:

$$U(n) = \nu(\mathtt{0},\ \mathtt{1},\ \mathtt{1},\qquad \mathtt{1})$$

where the number of 1's is n, and ν is a new varyadic functional symbol. Given t_1 and t_2, replace every number n by $U(n)$ in both terms, and denote the result as t_1', t_2'. Obviously, $n_1 \trianglelefteq n_2$ is true for the extended relation if and only if $U(n_1) \trianglelefteq U(n_2)$ is true for the original relation. Hence $t_1 \trianglelefteq t_2$ holds if and only if $t_1' \trianglelefteq t_2'$, which makes the extended HK theorem true. $\square$

3 Embedding on R-expressions

The syntax of data structures in Refal is defined as follows:

$$\begin{aligned} t &::= \mathtt{s}.i \mid \mathtt{t}.i \mid \mathtt{e}.i \mid s \mid n \mid (e) \\ e &::= \mathit{empty} \mid t\ e \end{aligned}$$

Here:

- t is in the set T_R of all *R-terms*;
- e is in the set E_R of all *R-expressions*;
- n is in the set N of all natural numbers;
- s is in a finite set S of all non-numerical symbols;
- $\mathtt{s}.i$, $\mathtt{t}.i$ and $\mathtt{e}.i$, where i is a numerical index, are three types of free variables: *s-variables*, *t-variables* and *e-variables*;
- round parentheses (...) are *structure brackets* to construct expressions;

$$\begin{aligned}
&M_{rt}[\mathtt{s}.i] = s_s\\
&M_{rt}[\mathtt{t}.i] = s_t\\
&M_{rt}[\mathtt{e}.i] = s_e\\
&M_{rt}[s] = s\\
&M_{rt}[n] = n\\
&M_{rt}[(e)] = f_{par}(M_{re}[e])\\
\\
&M_{re}[empty] = empty\\
&M_{re}[t] = M_{rt}[t]\\
&M_{re}[t_1\ t_2\ e] = M_{re}[t_1], M_{re}[t_2\ e]
\end{aligned}$$

Table 1: The mapping $M_{rt}\quad T_R \to T(F_R)$.

In our notation the variables like e (in italics) are elements of a metalanguage describing constructions of the language Refal. Thus e is a metavariable taking values from the domain E_R of all Refal expressions, while $\mathtt{e}.i$ is a free variable used in Refal expressions.

To use HK in the context of Refal, we map the set of all Refal terms T_R onto the domain of functional terms $T(F_R)$ over a certain set F_R of functional symbols:

$$F_R = S \cup N \cup \{s_s, s_t, s_e, f_{par}\}$$

where f_{par} is a varyadic functional symbol, while all the other symbols are constants.

The mapping $M_{rt}\quad T_R \to T(F_R)$ is recursively defined in Table 1

Example The R-term

$$t = \mathtt{((s.1\ s.2\ e.3)'ab'(25))}$$

is transformed into:

$$M_{rt}[t] = f_{par}(f_{par}(s_s, s_s, s_e), a, b, f_{par}(25))$$

where we presume that the English letters are among the symbols in S.

Now we simplify the notation of terms in $T(F_R)$. Since we have only one functional symbol in F_R, we can drop it, keeping in mind that each left parenthesis (stands for f_{par}(. We shall also replace commas that separate terms in expressions by blanks. The resulting term looks, and in fact is, an R-term, as long as s_s, s_t, s_e are included in the set S:

$$((s_s\, s_s\, s_e)\, a\, b\, (25))$$

We define the homeomorphic embedding relation $\trianglelefteq_R$ on R-terms T_R as the image of the relation $\trianglelefteq$ on $T(F_R)$:

$$t_1 \trianglelefteq_R t_2 \;\; \Leftrightarrow \;\; M_{rt}[t_1] \trianglelefteq M_{rt}[t_2]$$

To evaluate an instance of relation $\trianglelefteq_R$ on T_R we first use the mapping M_{rt} to $T(F_R)$ and then work in $T(F_R)$. As we have just seen, $T(F_R)$ differs, with our notation, from T_R only in that all free variables are replaced with constants. Therefore, we shall use the same symbol $\trianglelefteq_R$ for the embedding relation in both sets.

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

Definition 2 The homeomorphic embedding relation $\trianglelefteq_R$ on the set $T(F_R)$ holds in the following cases:

1. $s \trianglelefteq_R s$, where $s \in S \cup \{s_s, s_t, s_e\}$;
2. $n_1 \trianglelefteq_R n_2$, where $n_1, n_2 \in N$ and $n_1 \leq n_2$;
3. $t \trianglelefteq_R (e_1 t' e_2)$, where $t \trianglelefteq_R t'$ and e_i for $i = 1, 2$ are some expressions (this case to be referred as *term embedded as term*);
4. $(t_1 t_2 \ldots t_n) \trianglelefteq_R (e_1 t'_1 e_2 t'_2 \ldots e_n t'_n e_{n+1})$ where $t_i \trianglelefteq_R t'_i$ for $i = 1, \ldots, n$, and any of the expressions e_i may be empty (the case of *term embedded as expression*). $\square$

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

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

It is nice to know that an extra pair of parentheses on both terms does not change the embedding relation.

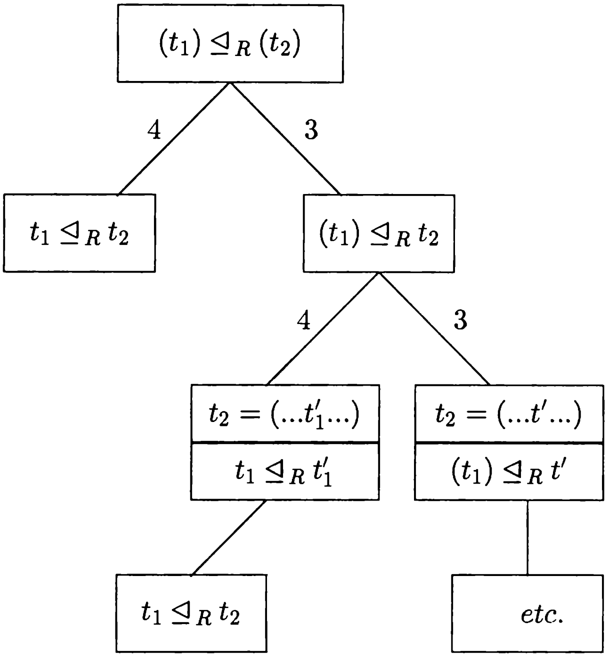
Theorem 3

$$(t_1) \trianglelefteq_R (t_2) \;\; \Leftrightarrow \;\; t_1 \trianglelefteq_R t_2$$

Proof If $t_1 \trianglelefteq_R t_2$, then by case 4 of Definition 2, $(t_1) \trianglelefteq_R (t_2)$. The converse is a bit more complicated. Let $(t_1) \trianglelefteq_R (t_2)$ be given (see Fig.1). This means that there is a validity demonstration in accordance with Definition 2. Consider the last step of this demonstration. It cannot be case 1 or 2 because of parentheses. Suppose it is case 4. Our goal, $(t_1) \trianglelefteq_R (t_2)$, is the following instance of the stated relation:

$$n = 1; t_1 = t_1; t'_1 = t_2; e_1 = e_2 = \textit{empty}$$

For this relation to be proven, the condition $t_1 \trianglelefteq_R t'_1$, which with our instantiation is $t_1 \trianglelefteq_R t_2$, must have been proven before; so it is true.



$(t_1) \trianglelefteq_R (t_2)$
4
3
$t_1 \trianglelefteq_R t_2$
$(t_1) \trianglelefteq_R t_2$
4
3
$t_2 = (...t'_1...)$
$t_1 \trianglelefteq_R t'_1$
$t_2 = (...t'...)$
$(t_1) \trianglelefteq_R t'$
$t_1 \trianglelefteq_R t_2$
etc.

Figure 1: Proof of Theorem 3

Suppose the last step was according to case 3. The instance is:

$$t = (t_1); t' = t_2; e_1 = e_2 = \textit{empty}$$

The relation that is assumed to be proven before is, after instatiation, $(t_1) \trianglelefteq_R t_2$. Now we have to consider the possible ways this relation at the one but last level of demonstration could have emerged (see Fig.1). As at the last level, this could have taken place through either case 4 or case 3.

In case 4 the instantiation is:

$$n = 1; t_1 = t_1; (e_1 t'_1 e_2) = t_2$$

The preproven relation, after instantiaiton, is $t_1 \trianglelefteq_R t'_1$, where t'_1 is one of the constituing terms of t_2. If this relation was proven then we derive, by case 3, that $t_1 \trianglelefteq_R t_2$ is also true.

The case 3 on the one but last level of demonstration is considered with the instantiation:

$$t = (t_1); (e_1 t' e_2) = t_2$$

The relation assumed to be proven is $(t_1) \trianglelefteq_R t'$ It is similar to the relation on the preceding level, but t_2 is replaced by t', which has at least one pair of parentheses less. Since removing parentheses from t_2 cannot go infinitely, only those cases remain where $t_1 \trianglelefteq_R t_2$. □

Definition 2 translates into the program in Table 3, which is written in a version of Refal. For a reader not familiar with the Refal notation the following comments will be sufficient.

A function call $f(arg)$ is written in Refal as <f arg>. The main format of a function definition is:

f $\{arg_1 = value_1;$

$\quad arg_n = value_n; \}$

where arg_i are patterns of the argument, and $value_i$ the corresponding patterns after one step of computation. Equations are tried in the order they are written. Variables in patterns are written without dots; their syntactic types are recognized by the first letter of the variable's name: `s` for symbol variables, `t` for term variables, and any of `e,x,y,z` for expression variables. We assume that natural numbers are used only in the format (`NAT` n). The part of a line starting with / is a comment.

As an extension of the basic format of function definitions, we allow to use auxiliary functions without giving them explicit names. The format of an equation can be:

arg_i , $expression$

$\quad\quad \{arg_1 = value_1;$

$\quad\quad\quad arg_m = value_m; \}$

When arg_i is matched succesfully, compute *expression* and start matching it to the left sides of equations, computing the nameless function defined by these equations, with *expression* as its argument.

Example Trace the evaluation of the relation:

$$(a(b)c) \trianglelefteq_R ((a)p(b)((c)))$$

Termination cases 1 and 2 are not applicable. Under case 3, function `Em` compares `t.1` $= (a(b)c)$ with each of the four terms in `t.2`: $(a), p, (b), ((c))$; none is embedding `t.1`. The case becomes 4 now; the constituents of `t.1` are compared one by one with the constituents of `t.2`, calling `Embr` recursively. $a \trianglelefteq_R (a)$ is successful under case 3; now we try to find embedding for (b) among the remaining terms. $(b) \trianglelefteq_R p$ does not work, so we try $(b) \trianglelefteq_R (b)$. Case 3 does not work, but case 4 requires $b \trianglelefteq_R b$, which is satisfied. Finally, $c \trianglelefteq_R ((c))$ is found to be true. after a double use of case 4. The answer: true

Definition 3 The *size* of an R-expression is computed according to these rules:

- $size[s] = size[n] = 1$;
- $size[(e)] = size[e] + 2$;
- $size[t_1\, t_2 \quad t_n] = size[t_1] + size[t_2] + \quad + size[t_n]$. □

Theorem 4 *The maximum run time for evaluation of the relation* $t \trianglelefteq_R t'$ *by the embedding algorithm on the set of R-terms is linear with* $size[t']$.

Proof by structural induction on the number of pairs of parentheses p in the term t' Obviously, the statement is true for the basis, $p = 0$: see the initial sentences in the definition of the function `Em`.

Our inductive assumption is that the theorem is true for t' with no more than p parentheses, i.e. there exists a constant c such that

$$Time[t \trianglelefteq_R t'] \leq c\, size[t']$$

for all such t'

Let $t' = (t'_1\, t'_2 \quad t'_n)$ be a term with $p+1$ pairs of parentheses. The maximal run time is the sum of the maximal times under the cases 3 and 4.

```
/ <Embr t1 t2> is the predicate  t1 ⊴_R t2
Embr {t1 t2 = <Dec t1 t2 <Em 3 t1 t2>>; }

/ sc in  Em is 3 when cases 3 and 4 of Definition 2
/ must be checked, and 4 when only case 4 remains
/ termination cases 1 and 2 are checked with any  sc
Em {
  sc s1 s1 = T;
  sc t1 s2    F;  a symbol embeds only itself
  sc (NAT s1) (NAT s2)   <Lesseq s1 s2> /  s1 ≤  s2
  sc t1 (NAT s2)    F; / a number may embed only a number
  sc ()(e2) = T; / empty is embedded in any expression
/  t2 = () signifies the end of a loop over terms in  e2
  3 t1 () = 4; / case 3 failed; try case 4 (see f-n  Dec)
  4 t1 ()    F; / case 4 failed; the answer is  F
/ a loop over terms in e2, case 3
  3 t1 (t2 e3), <Embr t1 t2>:
               {T = T;
                F = <Em 3 t1 (e3)>; };
/ a loop over terms in  e2 and  e1, case 4
  4 (t1 e2)(t1s e2s), <Embr t1 t1s>:
                          {T = <Em 4 (e2)(e2s)>;
                           F = <Em 4 (t1 e2)(e2s)>; };
    }

/ This function decides if case 4 must be considered
Dec { t1 t2 T = T;
      t1 t2 F = F;
      t1 t2 4 = <Em 4 t1 t2>; }
```

Table 2: A Refal program for the embedding relation $\trianglelefteq_R$.

In case 3 we compare t with the constituent terms t'_i, $i = 1, \ldots, n$. Since each of them has no more than p pairs of parentheses, the processing time for each t'_i will be no more than $c\, size[t'_i]$. In the worst case t will be compared with all of them. We assume that the maximal overhead for the treatment of the two parentheses which delimit the term is no greater than $2c$ (since this time is a constant, we can always choose a big enough c). So, we have:

$$Time[t \trianglelefteq_R t'] \leq \sum_{i=1}^{n} size[t'_i] + 2c = c\,(\sum_{i=1}^{n} size[t'_i] + 2) = c\, size[t']$$

In case 4 let $t = (t_1, t_2, \ldots, t_m)$. We start comparing t_1 with t'_i, where $i = 1, 2, \ldots, n$. If among those terms none is embedding t_1, it will pass to the end, examining each node t'_i, the same way as in case 3, and with the same result for the run time. If one of the terms in t', say t'_j, happens to embed t_1, then the past terms are ignored, and the next constituent term t_2 will be compared with the remaining terms, starting with t'_{j+1}. The process goes on like this, so that each constituent term of t' is examined no more than once. Therefore, we again have the same limit as in case 3, which proves the theorem. □

4 Embedding on lists

A *list* is a special kind of *s-expression.* An s-expression is defined as the set $T(F_L)$ of all functional terms constructed on the set F_L of all functional symbols which includes symbols from some finite set S (usually referred to as *atoms*), the constructor *cons* of arity two, and a special symbol *nil*. It can also be seen as a tree. A list is an s-expression which obeys the following restriction: the second argument y in the term $cons(x, y)$ is never an atom, but either a term $cons(x, y)$, or *nil*. A list may include variables. For the purpose of establishing the embedding relation we replace every variable with the symbol s_v, which we include in S. The set of such lists is denoted as T_L. As a simplification, we do not include numbers into T_L. If necessary, it is easy to do the way we did it for T_R.

The purpose of lists (as one can see from the word *list* itself) is to represent sequences, i.e. linear structures, not just binary trees, which they formally are. This goal is achieved by using *the list notation*, instead of the notation which is implied in their formal definition. In the list notation we write

$$(t_1\, t_2\, t_3 \qquad t_n)$$

to stand for

$$cons(t_1, cons(t_2, cons(t_3, \ldots cons(t_n, nil) \ldots)))$$

where each of the terms t_i must also be understood in this way.

The inverse transformation from the binary tree form into the list notation is done as follows: (1) delete all constructors *cons*; (2) delete all left parentheses that follow immediately after the comma, and erase their corresponding right parentheses; (3) delete all symbols *nil* (4) replace all commas by spaces.

Lists in the list notation look exactly the same as R-terms. Thus what in the list notation, say in a Lisp program, is written as:

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

is, actually, an element of $T(F_L)$:

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

To compare relations and algorithms on T_R and T_L, we shall represent lists by R-terms that look the same way in writing, but we shall treat them differently, keeping in mind that, the appearance notwithstanding, lists remain binary trees, not strings of terms we deal with in Refal. Refal expressions on their own are not allowed in T_L; they must always be kept inside some parentheses which delimit a list. There is no concatenation, as we know it in Refal. Lists are constructed and deconstructed on the left side only. Construction requires that the second operand always be a list proper, not an atom:

$$cons[t_1, (e_2)] = (t_1 e_2)$$

Deconstruction can be reduced to two basic functions:

$$head[(t_1 e_2)] = t_1; \qquad tail[(t_1 e_2)] = (e_2)$$

nil is represented as ().

Definition 4 The homeomorphic embedding relation $\trianglelefteq_L$ on the set of lists T_L is a special case of Definition 1 for $F = F_L$ In Refal representation it becomes:

1. $s \trianglelefteq_L s,$ if s is an atom;

2. $() \trianglelefteq_L ()$
3. $t \trianglelefteq_L (t'_1 e'_2)$ if $t \trianglelefteq_L t'_1$
4. $t \trianglelefteq_L (t'_1 e'_2)$ if $t \trianglelefteq_L (e'_2)$
5. $(t_1 e_2) \trianglelefteq_L (t'_1 e'_2)$ if $t_1 \trianglelefteq_L t'_1$ and $(e_2) \trianglelefteq_L (e'_2)$ □

The specialization is done as follows.

The constants of F_L are atoms and the special symbol *nil*, which results in statements 1 an 2. In the recursive part we have the only constructor *cons*, so $f = g = cons$ with the arity $m = 2$. The first case in Definition 1 produces, with $i = 1$,

$$t \trianglelefteq_L cons(t'_1, t'_2) \quad \text{if} \quad t \trianglelefteq_L t'_1$$

which is statement 3 in the Refal notation, and with $i = 2$,

$$t \trianglelefteq_L cons(t'_1, t'_2) \quad \text{if} \quad t \trianglelefteq_L t'_2$$

which is statement 4.

In the second recursive case of Definition 1 the constituent terms of the compared terms must be in relation $\trianglelefteq_L$ pairwise, because they are in the same number: $m = 2$. Hence:

$$cons(t_1, t_2) \trianglelefteq_L cons(t'_1, t'_2) \quad \text{if} \quad t_1 \trianglelefteq_L t'_1 \quad \text{and} \quad t_2 \trianglelefteq_L t'_2$$

This is nothing but our statement 5.

The relation $\trianglelefteq_L$ is similar to $\trianglelefteq_R$ in that a sequence of terms may be embedded in another sequence if all the terms are found there, even if separated by other terms, but still in the right order.

Example $(a\ b) \trianglelefteq_L (p\ a\ r\ b\ q)$ holds, as one can see from the following derivation:

$$\begin{array}{rcll}
() & \trianglelefteq_L & () & \text{case 2} \\
() & \trianglelefteq_L & (q) & \text{case 4} \\
(b) & \trianglelefteq_L & (b\ q) & \text{case 5} \\
(b) & \trianglelefteq_L & (r\ b\ q) & \text{case 4} \\
(a\ b) & \trianglelefteq_L & (a\ r\ b\ q) & \text{case 5} \\
(a\ b) & \trianglelefteq_L & (p\ a\ r\ b\ q) & \text{case 4}
\end{array}$$ □

However, these relations are not identical.

Example The following relation holds on T_L, but not on T_R:

```
Embl {
   s1 s1   T; / case 1
   t1 s2   F;
   ()(e1) = T; / case 2
   t1 ()   F;
   s1 (t2 e3), <Embl s1 t2>:
                 { T = T; / case 3 when t is s
                   F   <Embl s1 (e3)> };
   (t1 e2)(t1s e2s), <Embl (t1 e2) t1s>:
              {T = T; / case 3
               F, <Embl (t1 e2) (e2s)>:
                 {T = T; / case 4
                  F, <Embl t1 t1s>:   / case 5, check heads
                   {T = <Embl (e2)(e2s)>; / case 5, check tails
                    F = F} } } }
```

Table 3: Homeomorphic embedding on T_L; a recursive algorithm

$$(a\, b\, c) \trianglelefteq_L (a\, p\, (b\, c)\, q)$$

The derivation in T_L is:

$(b\, c)$	$\trianglelefteq_L$	$((b\, c))$	case 3
$(b\, c)$	$\trianglelefteq_L$	$((b\, c)\, q)$	case 3
$(b\, c)$	$\trianglelefteq_L$	$(p\, (b\, c)\, q)$	case 4
$(a\, b\, c)$	$\trianglelefteq_L$	$(a\, p\, (b\, c)\, q)$	case 5

Here parts of the left side are all on the same level, while in the right side they are distributed between two levels. It is easy to see that the relation $\trianglelefteq_R$ does not include such situations. Obviously, $(a\, b\, c)$ is not embedded as a term. Then we try to embed it as an expression. Term a is immediately found, and the problem now becomes to embed first b and then c (not the term $(b\, c)$, as in the case of lists!). Term b is embedded in $(b\, c)$, but then for c only q remains, so there can be no embedding. □

Definition 4 directly translates into the Refal program shown in Table 4.

Theorem 5 *The run time of the recursive algorithm for $t \trianglelefteq_L t'$ is, in the worst case, at least exponential with the size of the smaller operand:*

$$T_{worst}[t \trianglelefteq_L t'] > c\, 2^{min(size[t],size[t'])}$$

Proof Consider the case where both t and t' are lists of atoms, and let the number of atoms be m in t, and n in t' Let $T(m,n)$ be the worst run time for $t \trianglelefteq_L t'$ when m and n are given. There are four calls of the function *Embl*, and in the worst case all of them will be evaluated, hence

$$T(m,n) \geq T(m,1) + T(m,n-1) + T(1,1) + T(m-1,n-1)$$

The time $T(m,1)$ for the evaluation of `<Embr (t.1 e2)t1s>` where `t1s` is an atom, is constant (the result is always `F` in one step). $T(1,1)$ is also a constant: the time needed to compare two atoms. Hence we have:

$$T(m,n) > T(m,n-1) + T(m-1,n-1)$$

It can be seen that $T(m,n-1) \geq T(m-1,n-1)$. Indeed, compare the trees of recursive calls generated by the corresponding initial calls. Since we consider the worst case, the recursion will go on as long as each of the two arguments becomes 1 or 0. Therefore, $T(m,n-1)$ cannot generate less calls than $T(m-1,n-1)$, because it has the initial m which is greater by one than in $T(m-1,n-1)$.

Thus we have:

$$T(m,n) > 2\,T(m-1,n-1)$$

from which we derive that there exists such a constant c that

$$T(m,n) > c\, 2^{min(m,n)} \qquad \square$$

Because of its exponential time complexity, the recursive algorithm is hardly of any direct use. But using the method of dynamic programming (see, e.g., [1]) it can be converted into an iterative algorithm which is presented in [28]. The average run time of this algorithm is given by:

$$Time[(t_1 \ldots t_n) \trianglelefteq_R (t'_1 \,..\, t'_m)] = \tau \begin{cases} \mathcal{O}(m^2/2) & \text{if } m \leq n \\ \mathcal{O}(n(m-n/2)) & \text{if } m \geq n \end{cases}$$

where τ is the average time required for evaluation of $t \trianglelefteq_L t'$ by recursive calls of *Embl*.

5 Simulation whistles

A *whistle* is a relation on a set of terms T which is used in the *termination algorithm* as defined below. It can be used in supercompilation or elsewhere to terminate generation of a potentially infinite sequence of terms from T

Definition 5 The termination algorithm. Given a relation $whistle(t, t')$ on a set of terms T and a potentially infinite sequence of terms: $t_1, t_2, .$ to be referred to as a *T-process*, do:

```
        $i := 1;$
Loop1   $i := i + 1;$
        $j := i - 1;$
Loop2   $if\ whistle(t_j, t_i)\ then\ stop;$
        $j := j - 1;$
        $if\ j \geq 1\ then\ goto\ Loop2\ else\ goto\ Loop1$   □
```

Definition 6 A whistle is *unfailing* if the termination algorithm using it always stops. □

As we have proven above, the whistles $\trianglelefteq_R$ and $\trianglelefteq_L$ on their respective sets of terms are unfailing.

Theorem 6 *Let $W_1(t, t')$ be a whistle on a set of terms T_1. Let M $T_2 \to T_1$ be a mapping from some set of terms T_2 to T_1. Define a* simulation *whistle W_2 on T_2 as:*

$$W_2(t, t') = W_1(M(t), M(t'))$$

If the whistle W_1 is unfailing, then W_2 is also unfailing.

Proof Consider an arbitrary T_2-process. Define a parallel T_1-process by mapping each its term onto T_1 by M.

At every stage of the T_2-process the whistle $W_2(t, t')$ blows if and only if the whistle $W_1(M(t), M(t')$ blows. Since the T_1-process always stops, so does the T_2-process. □

We can use Theorem 6 to prove the acceptability of various non-trivial whistles on various sets of terms. We did it already when we defined embedding on Refal terms and expressions by reducing it to the relation on $T(F_R)$ by the mapping M_{rt}.

Here is another application of this theorem. Since there exists a mapping $M\quad T_L \to T_R$ from lists to R-terms, we can use the efficient unfailing whistle $\trianglelefteq_R$ to define the corresponding unfailing and efficient whistle on T_L. We just treat lists as R-terms. In this case the mapping is especially simple because in the list notation these objects look identical.

The sets T_1 and T_2 are not necessarily different. Mapping can take place within the same set of terms, but the resulting whistle may be different from the original one.

Definition 7 Let $W_1(t,t')$ and $W_2(t,t')$ be two whistles on a set T If $W_2(t,t')$ blows (i.e. is true) whenever $W_1(t,t')$ blows, and there exists at least one pair (t,t') such that $W_(t,t')$ blows, while $W_1(t,t')$ does not, we say that W_2 is *more eager* than W_1, or W_1 is *more conservative* than W_2. □

Theorem 6 opens a way to construction of unfailing whistles by mapping some basic embedding relation. If the mapping is many-to-one, the resulting whistles will be more eager than the original one.

Examples Assuming that $W_1(s,s) = true$, a simulation whistle which uses the mapping $M(t) = s$, where s is a fixed symbol, terminates the process at the second stage. Or consider a not so radical transformation. If we want, for the purpose of whistling, to make no distinction between symbols a and b, we can use a mapping that converts each b into a.

In the following example the functional symbol of a term becomes a part of the list of arguments. This makes the whistle different.

Let the set F of functional symbols include three constants: a, b, c and three varyadic symbols: f, g, u. Let $W_1(t,t') = t \trianglelefteq t'$ be an unfailing whistle on T(F). Define the mapping $M\quad T(F) \to T(F)$ where $M(s) = s$ for all constants s and

$$M(s_f(t_1, t_2, \dots, t_n)) = u(s_f(c), t_1, t_2, \dots, t_n)$$

for all functional symbols s_f. The whistle W_2, as defined in Theorem 6, is different from W_1. Indeed, let $t = f(a)$, and $t' = g(f(c), a, b)$. Then W_1, i.e.

$$f(a) \trianglelefteq g(f(c), a, b)$$

is false, while W_2, i.e.

$$u(f(c), a) \trianglelefteq u(g(c), f(c), a, b)$$

is true.

6 Generalization proper

With a given whistle algorithm, various algorithms of the generalization proper may be used. We do not present any specific and completed generalization algorithms in this paper. We only want to show some relationships between whistling at a situation of embedding and the subsequent generalization.

Definition 8 Let $t, t' \in T$, where T is T_R or T_L. We write $t \subseteq t'$ if there exists a substitution for the variables in t' that converts t' into t. We call a *generalization* of t and t', denoted as $gen(t, t')$, any such term $t^g \in T$ that $t \subseteq t^g$ and $t' \subseteq t^g$. □

First, let $T = T_R$. R-expressions (strings of terms) can be generalized keeping identical parts on both left and right sides.

Examples

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

Such, and more complicated, generalizations arise, in a natural way, from the algorithm that discovers the embedding relation. When a term is embedded as an expression, according to Definition 2,

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

where $t_i \trianglelefteq_R t'_i$ for $i = 1, \ldots, n$, we generalize the corresponding terms and replace non-empty expressions e_i by new free variables:

$$(x_1\, t^g_1\, x_2\, t^g_2 \quad . x_n\, t^g_n\, x_{n+1})$$

where $t^g_i = gen[t_i, t'_i]$.

Examples

$gen[(a\, b\, c\, c\, d\, m), (p\, a\, b\, c\, q\, r\, c\, d\, n)] = (x_1\, a\, b\, c\, x_2\, c\, d\, x_3)$
$gen[(a\, (b)\, c), (p\, a\, (b\, q)\, c) = (x_1\, a\, (b\, x_2)\, c)$ □

With $T = T_L$, i.e. the lists as the basic data structure, generalization can preserve only that common substructure which is on the left, but not on the right side. Even though a list, such as $(a\, b\, c)$, looks like a parenthesized string, it is, as we remember, a binary tree:

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

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

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

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

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

Unfortunately, when a program works by iterations (as opposed to recursive programs where data is passed from the value of one function to the argument of another) it is exactly on the left side that the lists are growing. The following example of supercompilation shows what are the results.

Example Suppose we want to find if a given string of symbols (atoms) contains a substring of four consecutive symbols which are either a or b. The method we choose is this: go through the string replacing every a by b; then go through it looking for $b\,b\,b\,b$. An iterative algorithm doing this job can be defined by recursive equations, Refal style, as follows:

$$f\,(x_1) = f_a\,([], x_1)$$

$$\begin{aligned} f_a\,(x_2, a\,x_1) &= f_a\,(b\,x_2, x_1) \\ f_a\,(x_2, s_1\,x_1) &= f_a\,(s_1\,x_2, x_1) \\ f_a\,(x_2, []) &= f_b\,(x_2) \end{aligned}$$

$$\begin{aligned} f_b\,(b\,b\,b\,b\,x_1) &= True \\ f_b\,(s_1\,x_1) &= f_b\,(x_1) \\ f_b\,([]) &= False \end{aligned}$$

This program can be understood as using either strings or lists. Concatenation of a term on the left side can be done and undone both on R-expressions, and on lists. [] stands for the empty expression or *nil*.

Function f_a, as it traverses the argument, reverses it: this is the only way available with lists (barring the use of *append* at each step).

Let us see how partial evaluation can be done by supercompilation. Let the initial call (*configuration*) be:

$F_1 \quad f\,(c\,a\,b\,a\,b\,d\,x_1)$

We expect that supercompilaiton will transform it into an identical *True.*

In our tracing of supercompilation below, each line represents a triplet in the process of the graph construction: a starting node, an edge, and the ending node. A colon after a node notation F_i signifies that the definition of the node (configuration) follows.

Driving F_1, we have an unconditional transition to F_2:

$F_1; F_2 \quad f_a\,([], c\,a\,b\,a\,b\,d\,x_1)$

Here the starting node is F_1, the edge is empty (no condiitons, no operations), and the ending node is F_2. Such configurations as F_1 are referred to as *transient*; they need not be kept in the graph constructed by driving.

The configuration F_2 is also transient, as are a few more, while function f_a processes the known part of the argument. Then:

$F_2; F_3 \quad f_a\,(c\,b\,b\,b\,b\,d, x_1)$

This configuration is not transient: there is a branching on x_1. We proceed driving along the first branch:

$F_3; x_1 \overset{c}{\rightarrow} a\,x_1; F_4 \quad f_a\,(b\,c\,b\,b\,b\,b\,d,\ x_1)$

where $x_1 \overset{c}{\rightarrow} a\,x_1$ is a *contraction*: a pattern-matching operation which separates the symbol a from the value of the variable x_1.

Configuration F_4 is not transient either, so we examine the history of driving looking for a possible need of generalization. We immediately find that $F_3 \trianglelefteq F_4$. Further development depends on the data structure we use. First, we consider the case of R-expressions:

$$gen[F_3, F_4] = f_a\,(x_2\,c\,b\,b\,b\,b\,d,\ x_1)$$

Now we redefine F_3 by reducing it to the generalized configuration which we still address as F_3:

$F_2; [] \overset{a}{\leftarrow} x_2; F_3 \quad f_a\,(x_2\,c\,b\,b\,b\,b\,d,\ x_1)$

Here the *assignment* $[] \overset{a}{\leftarrow} x_2$ is an operation which assigns to x_2 the value $[]$.

The processing of F_3 results in a finite graph whose meaning is a recursive program, the exit from which is a call F_4 of function f_b with a partially known argument:

$F_3; x_1 \xrightarrow{c} a\,x_1; b\,x_2 \xleftarrow{a} x_2;\ F_3$
$F_3; x_1 \xrightarrow{c} s_1\,x_1; s_1\,x_2 \xleftarrow{a} x_2;\ F_3$
$F_3; x_1 \xrightarrow{c} []; F_4 \quad f_b\,(x_2\,c\,b\,b\,b\,b\,d)$

Going on, we drive F_4, which results in the graph:

$F_4; x_2 \xrightarrow{c} b\,b\,b\,b\,x_2; True$
$F_4; x_2 \xrightarrow{c} s_1 x_2; F_4$
$F_4; x_2 \xrightarrow{c} []; F_5 \quad f_b\,(c\,b\,b\,b\,b\,d)$
$F_5; True$

In this graph there are three edges from the root F_4. One loops back to the root F_4, the other two return the value $True$. It is easy to have a supercompiler transforming F_4 into $True$ in this situation; the supercompiler described in [26] does it. After the substitution of $True$ for F_4 in the graph of F_3, this graph also comes to have a similar structure and will be replaced by $True$. Finally, the root F_1 of the whole graph becomes $True$. Partial evaluation is done in full. The new definition of the function f is:

$f(x_1) = True$

Now consider the case of the list data structure. Generalization of F_3 and F_4 yields $f_a\,(x_2)$, and the redefinition of F_3 is:

$F_2; c\,b\,b\,b\,b\,d \xleftarrow{a} x_2; F_3 \quad f_a\,(x_2,\ x_1)$

Now the configuration F_3 does not include the known part of the initial argument. The graph for it,

$F_3; x_1 \xrightarrow{c} a\,x_1; b\,x_2 \xleftarrow{a} x_2;\ F_3$
$F_3; x_1 \xrightarrow{c} x_1^s\,x_1; x_1^s\,x_2 \xleftarrow{a} x_2;\ F_3$
$F_3; x_1 \xrightarrow{c} []; F_4 \quad f_b\,(x_2)$

as well as the graph for F_4 it calls, treats function calls in their full generality. Partial evaluation has failed. It returned the original program. □

The reader could have noticed that the generalization algorithm for Refal data structure is given only for the case where the term is embedded as expression; the case of a term embedded as term was ignored. When a term is embedded as term we have the same problem as in the case of list structure: the only generalization is a free e-variable, e.g.

$$gen[a,\ (a)] = x_1$$

There is no way of keeping the common part, a in this example, in the generalization. This should be kept in mind when designing algorithms to be supercompiled. Parentheses must be used to separate static substructures, such as different arguments and subarguments of a function; dynamic accumulation of data should take place in the same substructure, as a growing string of terms. Such strings, however, may grow on different levels simultaneously, e.g.

$$gen[(a)(p),\ (a\,b)(p\,q)\,n] \qquad (a\,x_1)(p\,x_2)\,x_3$$

7 Termination of Supercompilation

HK alone does not immediately lead to the conclusion that the process of supercompilation is always finite. This would be true only if every walk (i.e. a path from the root to the current leaf) of the driving tree were growing monotonously. In fact, however, when a whistle $t_i \trianglelefteq t_j$ is given, the tree constructor returns and reduces either t_i, or t_j to the generalization $gen[t_i, t_j]$ (there are two versions of the supercompilation algorithm) Then the process is resumed starting from the generalization. To prove that supercompilation always terminates, we need to know that generalization of a term cannot be repeated infinitely.

Definition 9 Let $gen[t_1, t_2]$, where $t_1, t_2 \in T_R$ be a generalization function. Let $t_i \prec t_{i+1}$ denote that $t_{i+1} = gen[t_i, t'_i]$, where t'_i is some term, and $t'_i \not\sqsubseteq t_i$. Function gen is a *bounded* generalization if for every t_1 the sequence

$$t_1 \prec t_2 \prec t_3 \prec$$

can only be finite. □

Not every generalization function is bounded.

Example Let gen be such that

$$gen[(E), (E')] = (E\ e.i)$$

where E and E' are some Refal expressions (with variables, generally), and the variable $e.i$ has a *new* (i.e. not used in E) index i. Then the sequence

$$() \prec (e.1) \prec (e.1\ e.2) \prec (e.1\ e.2\ e.3)\ldots$$

goes on infinitely; hence this gen is not bounded. □

However, "reasonable" generalization algorithms tend to be bounded. In [24] a proof is given that algorithms resulting in a certain class of generalized expressions are all bounded.

Below we give a simple proof of termination in abstraction from many details of the algorithm of supercompilation; we only make a few assumptions about it.

Theorem 7 *The process of supercompilation using an unfailing whistle and a bounded generalization algorithm is finite.*

Proof Supercompilation is a controlled construction, through the use of driving, of a graph of configurations and transitions of the underlying computing machine. We assume that configurations, i.e. the nodes of the graph, are represented by the terms from some set T The graph is, essentially, a tree resulting from driving with two kinds of additions:

- A reduction of a configuration to its generalization, which does not violate the character of the graph as a tree.
- A reference from a node to one of its ancestors which indicates a reduction. It creates a cycle, but we shall consider such nodes as leaves of the tree, because so they are with regard to supercompilation process.

We assume that each new configuraiton C_i is first compared with its ancestors, in order to determine if it can be reduced to one of them. If it can, the reduction is made, and the current branch of the transition tree comes to an end. If there is no ancestor to reduce to, the whistle algorithm is called. If $whistle(t_k, t_i) = True$ for some ancester C_k, generalization is done as described above, and the generalized configuration is driven on.

We shall prove that under this assumptions every branch in the transition tree may only be finite. The order in which the graph is constructed, as well as other details of the algorithm remain arbitrary.

Let a *current walk* in a driving tree be a sequence of terms which starts with the root term, and such that if t_i is a term in the walk then the next term t_{i+1} is either one of the children of t_i or the child's generalization. Let a *historical walk* be an actual history of supercompilation which includes all nodes that have ever been constructed.

We define the following abstraction of the process of supercompilation. Two walks are constructed simultaneously: historical and current. Driving t_i and selecting a child t_{i+1} we add one more term to both historical and current walks. Returning from t_i to some t_k, where $k < i$, and generalizing it to t_k^1, we delete all terms succeeding t_k^1 in the current walk, but leave them in the historical walk. A return takes place every time when $whistle(t_k, t_i)$ is found to However be true, which means that no current walk will ever include such a pair of terms.

The construction of a walk ends when either the current t_i becomes passive (a constant, not a function call), or when it is looped back and reduced to one of its predecessors. In particular, when an attempt is made to generalize t_i with t_k which has the maximum of generality (the term $(e.k)$), any t_i is sure to be reduced to it; this will end the walk.

Since the out-degree of a node in the driving tree is bounded, we only have to prove that no historical walk in the tree is infinite.

$$\begin{array}{cccc}
t_1^{m_1} & t_2^{m_2} & t_3^{m_3} & \\
 & & & \\
t_1^3 & t_2^3 & t_3^3 & \\
t_1^2 & t_2^2 & t_3^2 & \\
t_1^1 & t_2^1 & t_3^1 & \\
t_1 & t_2 & t_3 & \bullet
\end{array}$$

Figure 2: Generalization columns

We call the *generalization column* of a term t_i the sequence of its generalizations:

$$t_i \prec t_i^1 \prec t_i^2 \prec \ldots t_i^{m_i}$$

which is always finite, because the generalization is assumed to be bounded. We assume, for simplicity, that at each level of generalization there is only one term; in fact there may be several of them, but this does not require any changes in our proof. Let us arrange the terms and their generalization columns as in Figure 2. If the current walk is finite (its end is shown in Figure 2 by $\bullet$), then the historical walk must also be finite, because at each return the level of generalization in one column increases by one. Taking the contrapositive, we conclude that if the historical walk is infinite, so is the current walk. However, if the current walk is infinite then we have an infinite sequence such that $whistle(t_k, t_i)$ does not hold for any pair k, i, where $k < i$. But this contradicts to the assumption that the whistle (see Definition 6) is unfailing. Therefore, every historical walk, and thus the process of supercompilation, is always finite. $\square$

In practice, supercompilers are *transient-skipping*, which means that no attempt is made to reduce or generalize a transient configuration. Obviously, Theorem 7 will not hold in the general case of transient-skipping supercompilation. It will hold, however, if computation by the original program always terminates.

Theorem 8 *If computation of the root node in supercompilation always terminates then, under the other assumptions of Theorem 7, the process of transient-skipping supercompilation is always finite.*

Proof We modify the proof of Theorem 7 as follows. Each transient node is left in both current, and historical walks. As before, the historical walk is finite if the current walk is finite, because the number of possible returns is still finite, and the total number of transient nodes is finite. By contraposition, if the historical walk is infinite then the current walk should be infinite. But it consists of two kinds of items: generalizable nodes and transient nodes. Both kinds must be finite in numbers: the former for the same reason as in Theorem 7, the latter because of computability of the initial configuration. This makes a contradiction. Hence the historical walk must be finite. □

References

[1] Cormen T.H., Leiserson C.E., Rivest R.L. *Introduction to Algorithms*, MIT Press, 1994.

[2] Dershowitz,N. Termination in rewriting, J. Symbolic Computation, 3, pp.69-116, 1987.

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

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

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

[6] Gallagher,J., Tutorial on Specialisation of Logic Programs, a tutorial, *PPEPM'93*, pp.88-98, ACM Press, 1993.

[7] R.Glück and A.V.Klimov. Occam's razor in metacomputation: the notion of a perfect process tree, in: P.Cousot, M.Falaschi, G.Filè, and Rauzy, ed. *Static Analysis*, LNCS vol.724, pp.112-123, Springer 1993.

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

[9] N.D.Jones, P.Sestoft and H.Sondergaard. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In: Jouannaud J.-P. (Ed.) *Rewriting Techniques and Applications*, Dijon, France, LNCS 202, pp.124-140, Springer, 1985.

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

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

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

[13] Komorowski, J., An Introduction to Partial Deduction, in: A.Pettorossi, Ed. *Proceedings META'92*, pp.49-69, LNCS 649, Springer, 1992.

[14] Leuschel, M. and Martens, B. Global Control for Partial Deduction through Characteristic Atoms and Global Trees, in: O.Danvy, R.Glueck and P.Thiemann Eds., *Partial Evaluation*, Intern. Seminar, Dagstuhl Castle, Germany, Febr. 1996, LNCS vol.1110, pp.263-283, Springer 1996.

[15] Lloyd, J.W and Shepherdson, J.C., Partial Evaluation in Logic Programming, *Journal of Logic Programming*, 11(3-4), pp.217-242, 1991.

[16] Martens, B. and Gallagher J., Insuring Global Termination of Partial Deductions while Allowing Flexible Polyvariance, in: L.Sterling, Ed. *ICLP'95* pp. 597-613, MIT Press, 1995.

[17] Nemytykh,A.P., Pinchuk, V.A., and Turchin,V.F., A Self-applicable Supercompiler, in: O.Danvy, R.Glueck and P.Thiemann Eds., *Partial Evaluation*, Intern. Seminar, Dagstuhl Castle, Germany, Febr. 1996, LNCS vol.1110, pp.322-337, Springer 1996.

[18] Nemytykh, A.P. and Pinchuk, V.A. Program Transformation with Metasystem Transitions: Experiments with a Supercompiler, in: D.Bjørner, M.Broy and I.Potossin Eds., Perspectives of System Informatics, 2-nd Internat. Andrei Ershov Conf. Novosibirsk, Russia, June 1996, Springer, LNCS vol.1181, pp. 249-261, Springer 1996.

[19] Pettorossi, A. and Proietti,M., A Comparative Revisitation of Some Program Transformation Techniques in: O.Danvy, R.Glueck and P.Thiemann Eds., *Partial Evaluation*, Intern. Seminar, Dagstuhl Castle, Germany, Febr. 1996, LNCS vol.1110, pp.355-385, Springer 1996.

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

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

[22] Sørensen, M.H. and Glück, R. An Algorithm of Generalization in Positive Supercompilation, in: Lloyd, J.W Ed., *Logic Programming: Proceedings of the 1995 International Symposium*, pp.465-479, 1995.

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

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

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

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

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

[28] Turchin, V.F., On Generalization in Supercompilation, *CCNY Technical Report*, 1996.

[29] Turchin,V.F. Metacomputation: Metasystem Transition Plus Supercompilation, pp. 481-510 in: O.Danvy, R.Glueck and P.Thiemann Eds. *Partial Evaluation*, Intern. Seminar, Dagstuhl Castle, Germany, Febr. 1996, LNCS vol.1110, pp. 481-510, Springer 1996.

[30] Turchin V.F., Supercompilation: Techinques and Results (invited talk) in: D.Bjørner, M.Broy and I.Potossin Eds., Perspectives of System Informatics, 2-nd Internat. Andrei Ershov Conf. Novosibirsk, Russia, June 1996, Springer, LNCS vol.1181, pp. 227-248, Springer 1996.

[31] Wadler, P., Deforestation: Transforming Programs to Eliminate Trees, *TCS*, vol.73, pp. 231-248, 1990.