A Note on Elimination of Simplest Recursions

Andrei P. Nemytykh 1,2
1 State Key Lab of Software Engineering
Wuhan University
Wuhan, Hubei, 430072 China
nemytykh@whu.edu.cn
2 Program Systems Institute RAS
Pereslavl-Zalessky
Yaroslavl region, 152140 Russia
nemytykh@math.botik.ru

ABSTRACT
Automatic program transformation, such as specialization, often introduces intermediate recursions that compute partial functions on parameters that are directly defined by simple recursive definitions. Replacing these recursive definitions by one-step definitions not only simplifies the residual program, but also allows more useful information to be propagated to remaining parts of the program, for continuing specialization.

We give an easily recognized characterization of a class of recursive definitions, called syntactical monomials, that can be transformed to one-step definitions. We show our transformation is meaningful for specialization of interpreters and self-application of specializers.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—Recursion, Control structures; F.3.2 [Logics and Meaning of Programs]: Semantics of Programming Languages—Partial evaluation, Program analysis

General Terms
Algorithms, Performance, Languages, Theory

Keywords
Automatic program transformation, Refal, Functional programming languages

1. INTRODUCTION
Residual simple recursions - and, more generally, intermediate simple recursions - are rather often produced by the program transformers with the goal of run time optimization. For example, both deforestation ([18]) and supercompilation ([16]) techniques will construct the simplest "empty" recursion as a result of transformation of the following program and its entry configuration:

\[
\text{repl}(B, A, \text{repl}(A, B, xS))
\]

where B and A are constants and

\[
\text{repl}(a, b, xS) = \text{if Null}(xS) \text{ then } [] \\
\text{else if Atom(car(xs)) then } \\
\quad \text{if } a =? \text{ car(xs) then } b : \text{repl}(a, b, cdr(xs))
\]

Such kind of a "non-realistic" definition, where the clause else car(xs); repl(a, b, cdr(xs)) is omitted, can appear as a result of the previous stage of specialization (see the second example given below and Section 7). A residual "empty" recursion will represent the identity function and will look as follows:

\[
\text{identity}(xS) = \text{if Null}(xS) \text{ then } [] \\
\text{else if Atom(car(xs)) then } \\
\quad \text{then car(xs) : identity(cdr(xs))}
\]

Specialization of the following second example by both generalized partial computation ([4]) and supercompilation ([16]) gives again a simple recursion:

\[
\text{replace}(a, a, xS)
\]

where both a and xS are parameters and

\[
\text{replace}(a, b, xS) = \text{if Null}(xS) \text{ then } [] \\
\text{else if Atom(car(xs)) then } \\
\quad \text{if } a =? \text{ car(xs) then } b : \text{replace}(a, b, cdr(xs)) \\
\quad \text{else car(xs) : replace(a, b, cdr(xs))} \\
\text{else replace(a, b, car(xs)) : replace(a, b, cdr(xs))}
\]

Here the residual recursion will look as follows:

\[
\text{project2}(a, xS) = \text{if Null}(xS) \text{ then } [] \\
\text{else if Atom(car(xs)) then } \\
\quad \text{if } a =? \text{ car(xs) then } a : \text{project2}(a, cdr(xs)) \\
\quad \text{else project2(a, car(xs)) : project2(a, cdr(xs))}
\]

Intermediate redundant recursions are serious hindrances for further sufficient transformations, because information of the recursive argument's structure is lost. The simplest residual recursions from these two examples set restrictions on the domains of the original programs. If a program transformer is allowed to extend the domains (as supercompilation, for example, does (see [16])), then transformations that
eliminate the above residual recursions result with perfectly simple programs: identity(xs) = xs and project2(a,xs) = xs respectively. Henceforth, we consider just transformers that are allowed to extend the function's domains.

This paper presents an algorithm for transforming programs that is capable of eliminating simple recursions defining the identity functions on the domains of the programs. A form of function definition specifying an identity partial function, called syntactical identity, that can be transformed into a one-step program\(^1\) is characterized. The algorithm is extended to multi-ary functions. In this case there exists a natural simple analogue of concept of the identity. The analogue is a monomial of concatenation. The corresponding form of function definition is called syntactical monomial.

Syntactical identities and concrete syntactical monomials of concatenation are easy to identify syntactically. However, a strategy is needed to create a hypothesis for the program to be a concrete monomial of concatenation (see Section 5). We suggest such strategy based on the unfolding procedure.

The characterization of the syntactic identities and the syntactic monomials is presented in a first-order strict functional language, where concatenation is associative and is used in infix notation. The associativity allows conveniently to formulate the characterization, but it can be reformulated in other terms (see an example in Section 8). The language has another constructor for composing tree structures. In fact, this language is just a dialect of Turán's Refal ([15]). So we will call it Refal as well with an index in order to distinguish its different versions.

Transformation by eliminating simple recursions is able to significantly decrease the time complexity of transformed programs. We show our transformation is meaningful for specialization of interpreters and self-application of specializers (Section 7).

The algorithm has partially been implemented as a tool of the supercompiler Scp4 (see [12],[11],[10]).

2. LANGUAGE REFAL\(_1\)

Refal\(_1\) language is a first-order strict functional language. It is a fragment of the basic Refal (see [15]) defined by:

```
program ::= SENTRY definition+
definition ::= function-name { sentence; } sentence ::= left-side = expression left-side ::= pattern expression ::= empty | term expression | function-call expression function-call ::= <function-name arg> arg ::= expression pattern ::= empty | term pattern term ::= SYMBOL | variable | (expression) variable ::= e.variable-name | s.variable-name empty ::= /* nilil */
```

Refal\(_1\) data are Refal data defined by the grammar:

```
d ::= (d\_1) | d\_1 d\_2 | SYMBOL | empty
```

Roughly speaking, a program in Refal\(_1\) is a term rewriting system. The semantics of the language 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 concatenation. It

\(^1\) A one line definition, whose right side has no function calls.

\(\textbf{SENTRY Go} \{\)
\[ e.xs = <\text{replace} 1 \ A \ e.xs > > ; \]
\[ \text{replace} \{\]
\[ s.a.b() = ; \]
\[ s.a.b(s.a.e.xs) = s.b <\text{replace} 1 \ s.a.b(e.xs)> ; \}
\(\textbf{SENTRY Go} \{
\[ s.a.e.xs = <\text{replace} 1 \ s.a.e.xs > > ; \]
\[ \text{replace} \{\]
\[ s.a.b() = ; \]
\[ s.a.b(s.a.e.xs) = s.b <\text{replace} 1 \ s.a.b(e.xs)> ; \]
\(\text{replace} \{\]
\[ s.a.b(s.a.e.xs) = s.x <\text{replace} 1 \ s.a.b(e.xs)> ; \]
\[ s.a.b((e.yz) e.xs) = (<\text{replace} 1 \ s.a.b(e.yz)>) \]
\[ <\text{replace} 1 \ s.a.b(e.xs)> ; \}
\(\textbf{SENTRY Go} \{
\]
```

\textbf{Figure 1: Example definitions}

\textbf{is binary, associative and is used in infix notation, which allows us to drop its parenthesis. The blank is used to denote concatenation. The second constructor is unary. It is syntactically denoted with its parenthesis only (that is without a name). The unary constructor is used for constructing tree structures. Every function is unary. Empty sequence is a special basic ground term. This term is denoted with nothing and called "empty expression". It is the neutral element (both left and right) of concatenation. All other basic ground terms are named as "symbols". There exist two types of variables - e.name and s.name. An e-variable can take any expression as its value, an s-variable can take any symbol as its value. For every sentence its set of variables from the left side includes its set of variables from the right side; there are no other restrictions on the variables. Associativity of concatenation may cause abstract pattern matching to be ambiguous on some patterns\(^2\). In the context of this paper, it is unimportant how the ambiguousness is concretely resolved. It is important that the pattern matching is done deterministic. The function-call name dictionary is a subset of the definition name dictionary.

Let a current active function call be given. A step of Refal\(_1\) machine is the following sequence of actions: pattern matching, replacement the right side variables with their values - with the result of the pattern matching, replacement of the active function call (in the function stack) with the updated right side and labeling of a new function call on the top of the changed stack as active.

The result of translation (into Refal\(_1\)) of the examples given in the introduction is shown in Figure 1 correspondingly. On the right hand side of the first sentences of the functions repl and replace we see the empty expressions. On the left hand side of these sentences the empty expressions are arguments of the parenthesis-constructors. Below we use sometimes the meta-symbol [] for the empty expression. The associativity of concatenation is shown, for instance, in

\(^2\) For example, the following equation e.1 e.2 = A B has three solutions: 1) e.1 = [] ; e.2 = A B; 2) e.1 = A, e.2 = B; 3) e.1 = A B, e.2 = [; In such cases the real Refal pattern matching takes the solution with minimal length of the datum taken by the first e-variable (from the left to the right) and so on by induction (see [12] for details). In our case the first solution e.1 = []; e.2 = A B will be chosen.
the both sides of the second sentence of the function repl. Consider a stepwise trace of a Refal computation for the second program of Figure 1. Let the computation start with the call Go A (\((A B)\)). Refal datum \((A B)\) represents a binary tree with the leaves \(A, B\). The computation proceeds with the following steps:

\[
\begin{align*}
2: & \langle \text{replace } A \ A \ (A \ B) \rangle \\
3: & \langle \text{replace } A \ A \ (A \ B) \rangle \\
4: & (A \langle \text{replace } A \ A \ (A \ B) \rangle \langle \text{replace } A \ A \ (A \ B) \rangle \\
5: & (A \ B \langle \text{replace } A \ A \ (A \ B) \rangle \langle \text{replace } A \ A \ (A \ B) \rangle \\
6: & (A \ B) \langle \text{replace } A \ A \ (A \ B) \rangle \\
7: & (A \ B)
\end{align*}
\]

The function append can be defined in Refal as:

\[
\text{ENTRY append } \{ (e.x) (e.y) = e.x e.y ; \}
\]

3. IDENTITIES

Definition 1. A Refal program is called a syntactical identity if and only if, after formal replacement of each function call \(\langle \text{function-name } arg \rangle\) from the program with its argument \(arg\), the left side of each sentence will be textually identical with the right side of the sentence.

Example 1. (A syntactical identity.)

\[
\text{ENTRY Go } \{ \ e.\text{input} = \langle \text{identity } e.\text{input} \rangle ; \}
\text{identity } \{ \\
\begin{align*}
\ &= ;; \\
\ &= s.1 \ e.x = s.1 \langle \text{identity } e.x \rangle ;
\end{align*}
\]

Statement 1. (A sufficient condition for identity.) Every definition from any syntactical identity Refal program defines a partial mapping \(F\) which is an identity on its domain: \(F(x) = x\).

Proof. Indeed, consider any \(x_0\) from the domain of \(F\). Then the call \(\langle F x_0 \rangle\) evaluates in a finite number of Refal steps \(n\).

By mathematical induction on \(n\). If \(n = 1\), then the right side of the corresponding sentence from the definition of \(F\) does not contain any function call and is equal to the left side. The base of the induction is proven.

Let \(n > 1\). Suppose the statement is proven for all definitions from the program and for all function calls from them, which evaluate in as many as \(m < n\) steps. Then the right side of a sentence, chosen by the pattern matching, contains at least one function call. Each function call \(\langle G \ arg \rangle\) from this right side evaluates in less than \(n\) steps and, therefore, by the inductive assumption, \(\langle [G \ arg] \rangle\) equals \(arg\).

Corollary 1. Every function call from any syntactical identity Refal program can be replaced with its argument by the program transformer allowed to extend the domains of the partial mappings defined by the programs to be transformed.

Proof. Immediately from Statement 1.

Example 2.

\[
\text{ENTRY Go } \{ \ e.\text{string} = \langle F e.\text{string} \rangle \} \\
\text{F } \{ \\
\begin{align*}
\ &= s.1 ; \\
\ &= s.1 \ e.\text{string} = \langle F s.1 \rangle \langle F e.\text{string} \rangle ; \\
\ &= s.1 ;
\end{align*}
\]

The residual program:

\[
\text{ENTRY Go } \{ \ e.\text{input} = e.\text{input} ; \}
\]

This example demonstrates that our transformation is able to decrease the time complexity from \(O(2^n)\) to \(O(1)\).

4. LANGUAGE Refal\(_N\)

Consider a superset of Refal\(_1\) by adding a possibility to define multi-ary functions. (Call the superset as Refal\(_n\).)

\[
\text{program := \$ENTRY definition+} \\
\text{definition ::= function-name \{ sentence; \} \} \\
\text{sentence ::= left-side = expression} \\
\text{left-side ::= patterns} \\
\text{expression ::= empty \ | \ term expression} \\
\text{term expression ::= function-call expression} \\
\text{function-call ::= \langle function-name arguments \rangle} \\
\text{arguments ::= expression \ | \ expression, arguments} \\
\text{patterns ::= pattern \ | \ pattern, patterns} \\
\text{pattern ::= empty \ | \ term pattern} \\
\text{term ::= SYMBOL \ | \ variable \ | \ (expression)} \\
\text{variable ::= e.variable-name \ | \ s.variable-name} \\
\text{empty ::= /\* nilhil */}
\]

Additional restrictions on the syntax: let a definition be given, 1) then all its sentences contain the same number of patterns and 2) the number of arguments in each function call \(\langle F \ arguments \rangle\) is equal to the number of patterns in the definition of \(F\).

Sets of variables from different patterns of a given sentence may intersect. The semantics of Refal\(_n\) programs is an obvious and natural extension of the semantics of Refal\(_1\) programs: each \(k\)-th argument from an active function call has to match the \(k\)-th pattern in the sentence (from the top to the bottom) in the active function’s definition.

5. MONOMIALS OF CONCATENATION

For multi-ary functions there exists a natural simple analogue of the concept of identity. Recall that Refal\(_n\) concatenation is associative.

Definition 1. Let a \(n\)-ary partial function \(F(x_1, x_2, \ldots, x_n)\) be given: \(DATA^n \rightarrow DATA\), where \(DATA\) is Refal\(_1\) data set. We say \(F\) is a partial monomial, if there exists a finite sequence of natural numbers \(k_1, k_2, \ldots, k_m\), such that

\[
F(x_1, x_2, \ldots, x_n) = x_1^{j_1} x_2^{j_2} \ldots x_n^{j_m}
\]

on the domain of \(F\), where \(0 < j_i \leq n\). (Here multiplication is concatenation, the power operator corresponds to the same constructor.)

Example 1. Copy \((x) = xx = x^2\). The copying function is a monomial of Refal\(_n\) concatenation.

Example 2. Concatenation \((x, y) = xy\). The concatenation function is a monomial of concatenation.

Example 3. The following program

\[
\text{ENTRY Go } \{ \ e.n1.1.2 = \langle \text{UnarySum } e.n1.1. e.n2 \rangle ; \}
\text{UnarySum } \{ \\
\begin{align*}
\ &= e.n1.1. \ I. e.n2 = I \langle \text{UnarySum } e.n1.1. e.n2 \rangle ; \\
\ &= e.n1.1. \ e.n2 = \langle \text{UnarySum } e.n1.1. e.n2 \rangle ;
\end{align*}
\}
\]

describes the partial monomial

\[
\langle \text{UnarySum } e.n1.1. e.n2 \rangle = e.n2 \ e.n1
\]

Example 4. The definition of Copy given below describes the partial monomial

\[
\langle \text{Copy } e.x \rangle = e.x \ e.x
\]
The second example definition given in Figure 1 describes a monomial: the projection on the second argument. The program append given in Section 2 defines the monomial from Example 2.

Definition 2. Let a *Refal* program $P$ be given such that the arities of all definitions from $P$ are the same and equal to $n$. Let $F(x_1, x_2, \ldots, x_n)$ be the input formats of the definitions from $P$. Let a finite sequence of natural numbers $k_1, k_2, \ldots, k_n$ be given, where $k_j > 0$ for all $j$. Denote $[y_j]$ for $y, \ldots, y_j$, where $y$ is repeated $j$ times. A formal arity raiser of the program $P$ (with respect to the $n$-tuple $k_1, k_2, \ldots, k_n$) is the following transformation of $P$: for every $i$

- each left side is $x_1^i, \ldots, x_n^i$, of each sentence from $P$ is transformed to the form $[x_1^{i-1}] \cdot \ldots \cdot [x_n^{i-1}]$;
- each function call $<F_j \ arg_1, \ldots, \ arg_n>$ from $P$ is transformed to the form $<F_j \ [arg_1]^{i-1}, \ldots, [arg_n]^{i-1}>$.

Denote the constructed program with $P^{k_1, \ldots, k_n}$ and we call it $k_1, \ldots, k_n$-ary with respect to $P$. (Definitions from $P^{k_1, \ldots, k_n}$ will be denoted with $F_i^{k_1, \ldots, k_n}$.)

Example 5. Consider the definition Double from Example 4. The formal raising of the arity of the definition with respect to the pair $2, 1$ yields the following program:

* Input formats: $\langle \text{Double}^{2,1}, \text{e}1, \text{e}2, \text{e}y \rangle$

\[
\begin{align*}
\text{ENTRY} \ \text{Double}^{2,1} \{ \\
\text{s.i.e.x, s.i.e.y, e.y} & = \text{e.s}1 \text{Double}^{2,1} \text{e.x, e.y s.s1}; \\
\text{.} & \text{. e.y} = \text{e.y}; \\
\text{\} }
\end{align*}
\]

STATEMENT 1. Let a *Refal* program $P$ be given and let $F_j$ be definitions from $P$. Let $P^{k_1, \ldots, k_n}$ be determined for a $n$-tuple $k_1, \ldots, k_n$. Consider new *Refal* definitions

\[
G_j \{ \text{e.s1, e.n} = \langle F_j \ [k_1]^{i-1}, \ldots, [k_n]^{i-1} \rangle; \\
\text{where each} \ F_j \ \text{is equal to} \ G_i, \ \text{for} \ j \ \text{and} \ i. \ \text{Let a function (described by the definition} \ F_j \ \text{is defined on a n-tuple} \ d_1, \ldots, d_n, \ \text{where each} \ d_j \ \text{is a Refal datum, then} \\
\langle \text{exists a} \ G_j \ d_1, \ldots, d_n \rangle \ \text{is equal to the number of Refal steps to evaluate} \ F_j d_1, \ldots, d_n > 1. \\
\text{And the number of Refal steps to evaluate} \\
G_j d_1, \ldots, d_n > \text{is equal to the number of Refal steps to evaluate} \\
F_j d_1, \ldots, d_n > \text{plus} \ 1.
\]

PROOF. By construction of the program $P^{k_1, \ldots, k_n}$ and by mathematical induction on the number of Refal steps evaluating the calls $F_j d_1, \ldots, d_n$.

The base: Let a call $F_j d_1, \ldots, d_n$ evaluates in one Refal step. Compared with the unary case, we did not impose any new conditions on choosing Refal sentences in the program $P^{k_1, \ldots, k_n}$, because equal data always come in places of the shared variables inside the repeated patterns from $P^{k_1, \ldots, k_n}$.

Hence, when the function call

\[
\langle F_j \ [d_1]^{i-1}, \ldots, [d_n]^{i-1} \rangle
\]

evaluates, then each sentence chosen by *Refal* pattern matching is the formal arity raiser’s image of the corresponding sentence from $P$ chosen during evaluation of the call $F_j d_1, \ldots, d_n$ from $P$ by *Refal* pattern matching. Additionally, the environments defined by these two pattern matchings are equal.

On the other hand, the right side of the sentence from $P^{k_1, \ldots, k_n}$ coincides with its pro-image, because the call $\langle F_j \ d_1, \ldots, d_n \rangle$, evaluates in one step and, therefore, the right side of the pro-image contains no function calls. Hence, the evaluated results of both calls are equal. The base of the induction is proven. Note we have just proved something more: let the value of the call $\langle F_j \ d_1, \ldots, d_n \rangle$ from $P$ be evaluated by one step, then the value of the call $\langle F_j \ [d_1]^{i-1}, \ldots, [d_n]^{i-1} \rangle$ from $P^{k_1, \ldots, k_n}$ is evaluated also by one step and these values are equal.

The step of the induction. Suppose the following statement is proven for each $i < m$: if the value of each function call $\langle F_j \ d_1, \ldots, d_n \rangle$ is evaluated by $i$ steps, then the value of the call $\langle F_j \ [d_1]^{i-1}, \ldots, [d_n]^{i-1} \rangle$ is also determined and is evaluated by $i$ steps.

Let the value of a function call $\langle F_j \ d_1, \ldots, d_n \rangle$ be determined and being evaluated with $m > 1$ steps, then literal repetition of the reasoning from the base case shows that *Refal* pattern matching at the first step of evaluation of the call $\langle F_j \ [d_1]^{i-1}, \ldots, [d_n]^{i-1} \rangle$ chooses a sentence-image (under the formal arity raiser) of the sentence from the program $P$ chosen by the first step of evaluation of the call $\langle F_j \ d_1, \ldots, d_n \rangle$, and the two environments after these pattern matchings are equal. Each function call from the right side of the pro-image evaluates less than $m$ steps. By the inductive assumption, the value of this function call coincides with the value of its image and takes the same number of steps to be evaluated.

Example 6. Get and Copy from Example 4 define the same partial mapping:

* ENTRY G \{ e.x = <Double\(^{2,1}\) e.x, e.x, e.x> \}

Definition 3. Let a *Refal* program $P$ be given such that the arities of all definitions from $P$ are equal one to another and equal to $n$. Let $F_1(x_1, x_2, \ldots, x_n)$ be the input formats of the definitions from $P$. Let a permutation $\sigma$ of the elements of the sequence $1, \ldots, n$ be given. The formal permutation $\sigma$ of the paramters of the program $P$ is the following transformation of $P$:

for every $i$

- each left side is $x_1^i, \ldots, x_n^i$, of each sentence from $P$ is transformed to the form $x_1^{\sigma(1)} \cdot \ldots \cdot x_n^{\sigma(n)}$;
- each function call $\langle F_i \ arg_1, \ldots, arg_n \rangle$ from $P$ is transformed to the form $\langle F_i \ arg_{\sigma(1)}, \ldots, arg_{\sigma(n)} \rangle$.

Denote the constructed program with $\mathcal{P}$ and call it $\sigma$-permuted with respect to $P$. (The definitions from $\mathcal{P}$ will be denoted with $F_j$.)

Example 7. Let $\sigma$ be the following permutation of the triple $1, 2, 3$: $\sigma(1) = 1, \sigma(2) = 3, \sigma(3) = 2$. Let $P$ be the program from Example 5. Then the program $\mathcal{P}$ is:

* Input format: $\langle \text{Double}^{2,1}, \text{e}1, \text{e}y, \text{e}2 \rangle$

\[
\begin{align*}
\text{ENTRY} \ \text{Double}^{2,1} \{ \\
\text{s.i.e.x, s.i.e.y, s.i.e} & = \text{s.s1} \text{Double}^{2,1} \text{e.x, e.y s.s1}; \\
\text{.} & \text{. e.y} = \text{e.y}; \\
\text{\} }
\end{align*}
\]
COROLLARY 1. Under the notations from Definition 3, the following semantic equality
\[ \langle F, d_1, \ldots, d_m \rangle = \langle F, d_1, \ldots, d, > \]
holds for all \( i \) and all \( \text{Refal} \) data \( d_1, \ldots, d_m \).

PROOF. Immediately from Definition 3. \( \square \)

Definition 4. We say a \textit{Refal} program \( P \) is a syntactical monomial, if and only if at least one exists which satisfies the following conditions: 1) the program \( P \) is a finite sequence of natural numbers \( k_1, \ldots, k_n \) such that the monomial \( P \) is determined; 2) there exists a permutation \( \sigma \) of the sequence \( 1, \ldots, n \) such that the formal replacement (in the program \( \sigma P \) )

- each function call \( \text{function-name } \arg_1, \ldots, \arg_m \) with concatenation of its arguments \( \text{arg}_{\sigma(1)}, \ldots, \text{arg}_{\sigma(m)} \)
- each left side = pattern \( \arg_{\sigma(1)}, \ldots, \arg_{\sigma(m)} \) of each sentence of \( \sigma P \) with concatenation of its pattern \( \text{pattern}_{\sigma(1)}, \ldots, \text{pattern}_{\sigma(m)} \),

the left side of each sentence from \( \sigma P \) will be textually identical with the right side of this sentence.

Example 8. The program from Example 5 is a syntactical monomial.

STATEMENT 2. \textit{(A sufficient condition for the monomiality)} Under the notations from Definition 3 and 4, let \( F_{j}(x_1, \ldots, x_n) \) be the input format of the definition \( F_j \). Let \( y_i \) be the \( i \)-th member of the sequence \( [x_1^i, \ldots, x_n^i] \), then \( F_j \) (for each \( j \) ) from the syntactical monomial \( P \) defines a partial mapping \( F_j \), which is the partial monomial:
\[ F_j(x_1, \ldots, x_n) = y_1, \ldots, y_n \]

PROOF. Literal repetition of the reasoning from the sufficient condition of identity (see Section 3) gives
\[ \langle F, x_1^i, \ldots, x_n^i \rangle \text{ y}_{\sigma(1)}, \ldots, y_{\sigma(m)} \rangle = \langle y_1, \ldots, y_n \rangle \].

Further, Corollary 1 (from the current section) yields
\[ \langle F, x_1^i, \ldots, x_n^i \rangle = \langle F, x_1, \ldots, x_n \rangle \text{ y}_{\sigma(1)}, \ldots, y_{\sigma(m)} \rangle \].

By Statement 1,
\[ \langle F, x_1, \ldots, x_n \rangle = \langle F, x_1^i, \ldots, x_n^i \rangle \text{ y}_{\sigma(1)}, \ldots, y_{\sigma(m)} \rangle \].

Hence, \( \langle F, x_1, \ldots, x_n \rangle = y_{\sigma(1)}, \ldots, y_{\sigma(m)} \). \( \square \)

We call the sequence \( k_1, \ldots, k_n \) together with the permutation \( \sigma \) satisfying the condition of Definition 4 (as well as the corresponding monomial \( y_{\sigma(1)}, \ldots, y_{\sigma(m)} \) ) a witness for syntactical monomiality of the program \( P \). We call a monomial \( x_1^{k_1}, \ldots, x_n^{k_n} \) a hypothesis of monomiality of the program \( P \), if the monomial is expected to be a witness for syntactical monomiality of this program.

COROLLARY 2. Under the notations from Statement 2, every function call \( F_j \text{ arg}_{\sigma(1)}, \ldots, \text{arg}_{\sigma(m)} \) from any syntactical monomial \( P \) can be replaced by the program transformer allowed to extend the domains of the partial mappings (defined by the programs to be transformed) with \( a_{\sigma(1)}, \ldots, a_{\sigma(m)} \), where \( a_i \) is the \( i \)-th member of the sequence \( \text{arg}_{\sigma(1)}, \ldots, \text{arg}_{\sigma(m)} \).

PROOF. Immediately from Statement 2. \( \square \)

Example 9. The following definition is a syntactical monomial and
\[ \langle \text{Double } e, e, y \rangle = e \cdot e \cdot e \cdot e \cdot e \text{ holds.} \]

\text{ENTRY} Double \{ s.e.x. e.y = s.1 \langle \text{Double } e, e, e, y, s.1 \rangle; e.y = e.y \}

5.1 The Strategy for Finding a Hypothesis of the Monomiality

The definitions and statements from the previous section give a simple algorithm to check whether a \textit{Refal} program \( P \) is a syntactical monomial or not. If the sufficient condition holds, then the program \( P \) can be transformed to a one-step program. A necessity arises to construct a hypothesis of the monomiality or to reject the program to be a syntactical monomial. We suggest to construct such a hypothesis on the basis of the formal exits from the recursion defined with the program \( P \) being analyzed. That is to say, on the sentences from the program, the right sides of which contain no function calls.

Let a \textit{Refal} program contain only the definition \( F \) and let \( F \langle x_1, \ldots, x_n \rangle > \) be its input format. Consider all possible sentences from \( F \): \( p_1, \ldots, p_{m_n} = \text{right side}_i \). Let \( d_{i_1} \) denote given fixed data. Suppose for every \( i \) there exists at least one \( n \)-tuple \( d_{i_1}, \ldots, d_{i_n} \) such that the definition \( F \) describes a partial function, such that in the first step of evaluation of \( F \langle d_{i_1}, \ldots, d_{i_n} \rangle > \) the pattern matching chooses a sentence \( p_1, \ldots, p_{m_n} = \text{right side}_i \) as a successful one. In the other words, the \( n \)-tuple is one of the basic cases of the recursive definition of \( F \). The \textit{right side}_i, by definition, is not changed during the checking on a syntactical monomial. Hence, if \( P \) is a syntactical monomial, then there exists a monomial \( M_i(x_1, \ldots, x_n) \) such that the result of the replacement of the variables \( x_1, \ldots, x_n \) with the corresponding patterns \( p_1, \ldots, p_{m_n} \) is \( \text{right side}_i \). For every \( i \) and \( j \) the following textual equality \( M_i(x_1, \ldots, x_n) = M_j(x_1, \ldots, x_n) \) has to hold.

The algorithm:

- For each passive sentence \( p_1, \ldots, p_{m_n} = \text{right side}_i \) let \( k_m (1 \leq m \leq s) \) stand for the ordinal numbers of the patterns in the left side of the sentence, whose lengths are greater than zero, and let \( t_l \) stand for the ordinal numbers of the patterns, whose lengths are zero. Construct a set of the monomials
\[ \text{set} = \text{Monom}(p_{t_1}(x_1), \ldots, p_{t_s}(x_s), \text{right side}_i) \].

The set of the hypotheses (the monomials) for the given sentence is
\[ \{ y_1, \ldots, h_{i-1}, h_i, y_j \} \mid y_1, \ldots, y_j \in \text{set} \}, \quad h_i \in \text{set}, \quad 0 \leq l < j \}, \]
where \( M \) is a set of the monomials defined by the following grammar: \( M ::= x_1 \text{ M } | \text{ empty} \). Here empty is the empty string (null).

- The set of the hypotheses for the given definition \( F \) is the intersection of \( \text{set} \) for all \( i \).

Where the function \text{Monom} is

* Input Format: \text{Monom}(\text{patterns}, \text{string})

\begin{verbatim}
Monom { if length(string) \neq 0, then for(every p_{i+m}, such that (string = p_{i+m},tail)) set_{i+m} = \{x_{i+m}, h \mid h \in \text{Monom}(\text{patterns}, \text{tail})\}; return(\text{The union of set}_{i+m} for all m}); else return(The empty set);} }
\end{verbatim}
The described strategy (see above) for generating the hypotheses has a drawback: a number of hypotheses can be constructed for the given definition F.

Example 1. Consider Example 9 from Section 5. The only exit from the recursion (the second sentence) allows us to generate an infinite number of hypotheses of the form e.x^m e.y e.x^m (k, m ∈ N). The pattern, whose length is zero, causes that.

Finiteness of the lengths of all right sides immediately gives:

**STATEMENT 1.** Under the notations from this section (see above), let a program contain at least one sentence

p_1, ..., p_m = right side, such that the right side does not have function calls and let each pattern p_i be a non-empty expression. Then there can exist only a finite number of hypotheses, which can be generated by the above-described strategy.

Example 2. The following definition F is a syntactical monomial. The syntactical monomial represents both

s.e.y e.s and s.e.y e.s.

* Input Format ⟨F s.x, e.y, s.z⟩

$ENTRY F \{ \ A, e.y, A = A . e.y e.y \ A; \} \)

Sharing syntactical entities with different patterns from a left side of a sentence causes the existence of a number of hypotheses. From our point of view, the sentences with such property are encountered inefinitely and the algorithm may consider all hypotheses or take into account only the one generated first.

By the definition of the data (see Section 2), the empty expression in a left side is a natural exit from a recursion and this case is appearing very frequently. The more passive sentences in a program, the lesser variants to generate hypotheses of the monomality.

Example 3.

* Input Format ⟨G e.x, e.y⟩

$ENTRY G \{ \[] . A . e.y = A < G \[\] , e.y; \[] . s.z . e.y = s.z e.y; \}

The definition G is a syntactical monomial. Recall [] stands for the empty expression. The first passive sentence generates a set of the hypotheses of the form e.x^k e.y e.x^l, while the second passive sentence generates a set of the hypotheses of the form e.x^k e.y e.x^l (k, l ∈ N). The intersection of these two sets contains two monomials. The syntactical monomial renders exactly the two monomials e.x e.x and e.x e.y.

Additional syntactical basic cases of an inductive definition can be obtained with the unfolding of the right sides of the recursive sentences from the program F (see []).

Example 4. After unfolding Example 9 from Section 5 we have the following program:

$ENTRY Double1 \{ \}

s.1 s.2 e.x e.y = s.1 s.2 \langle Double1 e.x, e.y s.1 s.2 \rangle;

s.1 . e.y = s.1 e.y e.s.1;

e.y = e.y; \}

The second sentence of which immediately yields the only correct hypothesis.

Though different hypotheses render the same partial function, they are able to lead to different transformations by further tools of a program transformer.

Example 5.

* Input Format ⟨Concat e.x, e.y⟩

$ENTRY Concat \{ s.z . e.y = G e.y, s.z; \}

s.z e.x . e.y = s.z \langle Concat e.x, e.y \rangle;

Where G is defined in Example 3 from the current section. The first hypothesis (see Example 3) allows to simplify of the definition Concat (by our algorithm) to

$ENTRY Concat \{ e.x . e.y = e.x e.y \}; \}

while the second hypothesis does not.

6. PARTIAL REFAL, EXPRESSIONS

Our restriction (see Definition 2 from Section 5) on the degrees k_j (to be positive) of monomials x_1^{k_1} ... x_n^{k_n} of concatenation can be easily canceled. I.e. it is possible to allow zero as a value of the degrees. Let k_j = 0 for some j, then the formal arity "raising" with respect to the variable x_j will become a formal arity reducing: that is the deleting of this formal variable and the patterns, which correspond to x_j , in the program should be transformed.

Example 1.

* Input Format ⟨F e.x, e.y⟩

$ENTRY F \{ \}

s.1 e.x = s.1 F^{1.0} e.x;

e.x = e.x; \}

A partial monomial F(x_1, ..., x_n) is called a partial projection, if only if, there exists a number j, such that 0 < j ≤ n and F(x_1, ..., x_n) = y. The projections are pretty important specific cases of the monomials. The example given above represents a projection on the first argument. Specialization by the supercompiler Sep4 ([12], [10], [7], [8]) of the second program of Figure 1 yields another example of a syntactical projection:

$ENTRY Go \{ s.a e.x = \langle R s.a.e.xs \rangle; \}

R \{ s.a . \[] = ; \}

s.a . s.a e.xs = s.a \langle R s.a.e.xs \rangle;

s.a . s.e.xs = s.x \langle R s.a.e.xs \rangle;

s.a . (e.xs) e.xs = \langle R s.a.e.xs \rangle \langle R s.a.e.xs \rangle; \}

Further this program is transformed to

$ENTRY Go \{ s.a e.xs = e.xs \}; \}

Moreover, the algorithm can be applied to verify not only the syntactical monomality but also a syntactical one-step property of Refal programs.

Definition 1. Let a n-ary partial function F(x_1, ..., x_n) be given: DATA^n → DATA^1, where DATA is Refal data set. Let
\[ R \text{-} \text{expr}(x_1, \ldots, x_n) := R \text{-} \text{term}(x_1, \ldots, x_n) \mid \text{empty} \]

\[ R \text{-} \text{term}(x_1, \ldots, x_n) := \text{SYMBOL} \mid \text{variable} \mid (R \text{-} \text{expr}(x_1, \ldots, x_n)) \]

\[ \text{variable} ::= x_1 \mid \ldots \mid x_n \]

\[ \text{empty} ::= [] \]

We say \( F \) is a partial expression, if and only if there exists a \( R \text{-} \text{expr}(x_1, \ldots, x_n) \) such that the following equality \( F(x_1, \ldots, x_n) = R \text{-} \text{expr}(x_1, \ldots, x_n) \) holds on the domain of the partial function \( F \).

Note, \( x_1, \ldots, x_n \) denote arbitrary abstract formal variables (but not only \( \text{Refal} \), variables). Under these notations, each pattern and any passive piece of each sentence from a \( \text{Refal} \) program \( P \) represent monomials of concatenation, when we will treat all their basic constants as abstract formal variables. In particular, the left structure bracket ( \( ( \) is treated as a self-sufficient syntactical entity: an abstract formal variable, whose name is ( Similarly for the right structure bracket )). If we will allow such extension of the set of the “variables” (which is described above) and formally extend the input formats (and correspondingly the left sides), then literal repetition of all definitions and statements from Section 5 gives an algorithm to verify whether a \( \text{Refal} \) program with the extended set of the variables is a syntactical monomial (and, hence, whether the program is a syntactical \( \text{Refal} \) expression).

\textbf{Example 2.}

\begin{verbatim}
* Input Format <F e.x. e.y>
ENTRY F {
e.y. s1.e.x = <F e.y. s1. e.x.;
e.y. e.x = (e.y. e.x (A));
}

After the formal extension of the input format with the basic constant, we have:

\begin{verbatim}
* Input Format <FF e.x. e.y. ( ). A>
ENTRY FF {
e.y. s1.e.x.(.), A= <FF e.y. s1. e.x.(.), A;= (e.y. e.x (A));
}

A single hypothesis of the monomiality is defined with the second sentence. After rising the arity and a suitable permutation of the formal variables we obtain the syntactical monomial (e.y. e.x (A)) of the following formal variables e.x. e.y. ( ). A, i.e. a \( \text{Refal} \) expression.

\begin{verbatim}
* InputFormat <G e.x. e.y>
ENTRY G{e.x. e.y = <FF ( e.y. e.x. ( . A. ));}

* Input Format <FF ( . e.y. e.x. ( . A. ));> FF {
 ( . e.y. s1.e.x. ( . A. ));
 ( . e.y. e.x. ( . A. )); = (e.y. e.x (A));

Moreover, we can replace the right side of the second sentence of the definition \( F \) with a functional call \( \langle G \text{ (e.y. e.x (A))} \rangle \) and treat the function call constructor as some new formal syntactical variables. Similar manipulations prove the definition \( F \) is a formal syntactical monomial representing \( \langle G \text{ (e.y. e.x (A))} \rangle \). As a consequence, we can replace any call \( \langle e \text{ (x, y)} \rangle \) with the call \( \langle G \text{ (e.y. e.x (A))} \rangle \).
\end{verbatim}
\end{verbatim}

\textbf{Definition 2.} Let a \( n \)-ary partial function \( F(x_1, \ldots, x_n) \) be given: \( DATA^* \rightarrow DATA_1 \), where \( DATA \) is \( \text{Refal} \) data set. We say \( F \) is a quasi-monomial, if and only if there exist two partial monomials \( M_1(x_1, \ldots, x_n) \), \( M_2(x_1, \ldots, x_n) \) on \( DATA^* \) and a partial function \( g: DATA^* \rightarrow DATA_1 \), such that the following equality \( F(x_1, \ldots, x_n) = M_1(x_1, \ldots, x_n) \mid g(x_1, \ldots, x_n) \mid M_2(x_1, \ldots, x_n) \) holds on the domain of the partial function \( F \).

The algorithm given in Section 5 can be easily extended to such quasi-monomials. We demonstrate the idea with:

\textbf{Example 3.}

\begin{verbatim}
ENTRY Go {
e1.n.e2 = <UnarySumMult e1.n1 [ ] . e2.n2;}

* Input Format <UnarySumMult e1.n.e.m.e2>
UnarySumMult {
 I.e1.n.e.m.e2
 = I <UnarySumMult e1.n1.e.m.e2.n2;[]
 , e.m.e2 = e.m.e2.n2;}

This program calculates the mathematical expression \( n_1 + n_2 \).
Let a hypothesis be given that \( \text{UnarySumMult} \) renders the quasi-monomial \( e.g.(e1.n1.e.m.e2.n2) \). The hypothesis can be generated by a variation of the method described in Section 5.1. Formally raise the arity to

\begin{verbatim}
* Input Format <SumMultI e1.n1.e1.n1.e.m.e2.e2>n2>
SumMultI {
 I.e1.n1.I.e1.n1.e.m.e2.e2.n2
 = I <SumMultI e1.n1.e1.n1.e.m.e2.n2.e2.n2;[]
 , e.m.e2.e2.n2 = e.m.e2.n2;}

Matching the right side of the passive sentence with the hypothesis yields that \( e.m \) is evaluated by \( g \). Formally replace the original sequence of the arguments, the original sequence of the patterns and the expression \( e.m \) (in the right side of the second sentence) corresponding to \( g \) with the formal variable \( e.m \).

\begin{verbatim}
* Input Format <SumMult2 e1.n1.e.formal.e.m.e2>
SumMult2 {
 I.e1.n1.e.formal.e.m2
 = I <SumMult2 e1.n1.e.formal.e.m2.n2;[]
 , e.formal.e2 = e.formal.e2.n2;}

Futher, the definition \( \text{SumMult2} \) is the syntactical monomial \( e.g.(e1.n1.e.formal.e2.n2) \). We can replace the original program with

\begin{verbatim}
ENTRY Go {
e1.n2 = e1.n2 e1.g(e1.n1 [] . e2.n2;}

* Input Format <g e1.n1.e.m.e2>
g {
 I.e1.n1.e.m.e2 = <g e1.n1.e.m.e2.n2;[]
 , e.m.e2 = e.m;}

Though the recursion was not eliminated, the values of the parameters \( e1.n1 \) and \( e2.n2 \) were taken out from the functional call \( \text{UnarySumMult} e1.n1 [ ] . e2.n2 \) and, hence, information of the structure of the arguments is not lost.
7. ONE MORE EXAMPLE

In the author's opinion, the task of describing simple
classes of programs (the systems of recursive equations) al-
lowing some simpler specifications of the partial functions
defined by these programs is interesting itself. The identity
functions and the projections play a basic part in the theory
of recursive functions ([9]).

In the point of view of program transformation, any triv-
ial change of intermediate recursions constructed by the
transformers is very desirable. That not only simplifies the
residual program, in which the recursions may be inserted,
but and provides additional information for specialization
of still unprocessed parts of the program to be transformed.
One cannot expect simple redundant recursions in the or-
iginal program written by a human, but the automatic spe-
cializers rather often produce such recursions. The problem
of simplification of the partial functions being the mono-
nial of concatenation arose as a result of analyzing residual
programs constructed by the supercompiler.

To illustrate a situation where syntactical monomial rec-
ursions arise, we consider self-interpreter (self-application
in Refal terms) and use the following mapping for represen-
ting Refal programs and Refal data with Refal data. Denote the mapping by underlining:

\[ \text{expr} = \text{expr} \]
\[ \text{symbol} = \text{symbol} \]
\[ [] = [] \]
\[ \text{expr}_1, \text{expr}_2 = \text{expr}_1 \text{expr}_2 \]

Let the following classical task of specializing an inter-
preter \texttt{IntProg} program \texttt{<IntProg.e.d>}
be given, where \texttt{e.d} stands for unknown (dynamic) data. Then the range of
the parameter \texttt{e.d} coincides with the image of Refal data
under the encoding function. To improve the specialization
results the user can use simple syntactical identity functions
to add annotations that constrain the values of the unknown
recursive data. On the one hand the specialist may use the
information indicated in the annotations, on the other hand
the residual recursions generated by these annotations will
be automatically eliminated from the residual program, if
the given dynamic typing is not used by the specialist.

Consider a function for substitution of the value of a vari-
able (of an environment) in a given expression. The function
is a simplified version of one of the critical functions from
any interpreter and any specialist.

* Input Format: \texttt{<Subst ((Var t.s n.) e.v. e.expr>}

\[ \text{Subst} \{ \]
\[ ((\text{Var} \ t.s \ n.) \ e.v) \ (\text{Var} \ t.s \ n.) \ e.expr = \text{e.v Subst ((Var} \ t.s \ n.) \ e.v) \ e.expr; \]
\[ ((\text{Var} \ t.s \ n.) \ e.v) \ (\text{'*'} \ e.expr1) \ e.expr = \text{'*'} \ Subst ((\text{Var} \ t.s \ n.) \ e.v) \ e.expr1; \]
\[ \text{Subst ((Var} \ t.s \ n.) \ e.v) \ e.expr; \]
\[ ((\text{Var} \ t.s \ n.) \ e.v) = ; \}

During interpretation the value of \texttt{e.d} (or its piece) be-
comes an argument of the function \texttt{Subst}:

\[ \text{Subst ((Var} \ t.s \ n.) \ e.v) \ e.d. \]

Thus the specialist has to specialize the following task
\[ \text{Subst ((Var} \ t.s \ n.) \ e.v) \ e.d> \] (as an intermediate
task), where \texttt{e.v} can be static or dynamic. We may annotate
the fact of the restriction on the range of \texttt{e.d} by means of the
composition: \texttt{<Subst ((Var t.s n.) e.v.) \text{'}Filter e.d>},
where the partial function \texttt{Filter} is defined below.

\[ \text{Filter} \{ \]
\[ (\text{'*'} \ e.y.s) \ e.x.s = (\text{'*'} \text{'}\text{'}\text{'} \text{'}Filter e.y.s) \text{'}Filter e.x.s; \]
\[ \text{e.s.a e.d} = \text{e.s} \text{'}\text{'}\text{'}\text{'}\text{'}<Subst ((Var t.s n.) e.v.) \text{'}Filter e.d> ; \]
\[ = ; \}

The intermediate task of specializing looks as follows:

\[ \text{Subst ((Var} \ t.s \ n.) \ e.v.) \text{'}\text{'}\text{'}\text{'}\text{'}\text{'}<Filter} \text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{'}\text{
als of concatenation. The algorithm is able to significantly decrease the time complexity of transformed programs.

We have shown that using our transformation is meaningful for specialization of interpreters and self-application of specializers.

This algorithm has the widest application to "tail-recursive" programs. Our experience shows that, after simplification of the compositional structure of a program by other tools of the supercompiler Scp4 ([12], [10], [7], [8]), the program frequently has the "tail-recursive" property.

The associativity of concatenation allows convenient formulation of our algorithm. But it is straightforward to reformulate our result in other, more widely used, functional languages. (For example in the ML language, the definitions of the syntactical identity (see Section 3) and the syntactical projection (see Section 6) can be used without any changes modulo the denotations. In the case of the general syntactical monomial, we have to take into account the correlation between the cons operator ; and the append operator \( \circ \).) Consider the following example from [18]:

\[
\text{tree } \alpha ::= \text{Leaf } \alpha \mid \text{Branch( tree } \alpha \text{) (tree } \alpha \text{)}
\]

\[
\text{flip : } \text{tree } \alpha \rightarrow \text{tree } \alpha
\]

\[
\text{flip } z t = \text{case } z t \text{ of }
\]

- \text{Leaf } z : \text{Leaf } z
- \text{Branch } x t y t : \text{Branch (flip } y t \text{) (flip } x t \text{)}

Applying the deforestation algorithm to \text{flip (flip } z t \text{)} yields:

\[
\text{h } z \text{ where }
\]

\[
\text{h } z = \text{case } z t \text{ of }
\]

- \text{Leaf } z : \text{Leaf } z
- \text{Branch } x t y t : \text{Branch (h } x t \text{) (h } y t \text{)}

After erasing the function application brackets and the function names inside these brackets in accordance with algorithm given in Section 3, both patterns are textually identical with the right sides of the corresponding sentences. Thus the definition \( \text{h} \) describes a partial identity function.

And finally, consider the second example from [18] in Refal, terms:

\[
\begin{align*}
\text{ENTRY append0} & \{ \\
& \text{[ } e. y = e. y ; \\
& (e.1) \text{ e. e. y } = (e.1) \text{ append0 e. e. y. ; } \\
& \}
\end{align*}
\]

Applying the supercompiler Scp4 (partially using our algorithm) to \text{append0} \text{ append0 e. e. y. e. z} gives the following one-step residual program:

\[
\text{ENTRY App } \{ \text{ e. e. y. e. z } = \text{ e. e. y. e. z. } \}
\]

9. ACKNOWLEDGMENTS

This work could not have been completed without support, permanent attention of Prof. V.F. Turchin. I am grateful to Wuhan University (China) and Prof. He Keqing for partial support of the work. The author thanks A.P. Konyshhev and A.V. Korlyukov for useful discussions of my work and interest to it. I am thankful to K.V. Shchakho for reading the paper.

The author wishes to thank the anonymous reviewers, who provided many valuable suggestions. I am also grateful to Carolyn Talcott for assistance in revising the paper.

10. REFERENCES


