ARITY RAISER AND ITS USE IN PROGRAM SPECIALIZATION

Sergei A.Romanenko

Keldysh Institute of Applied Mathematics
Academy of Sciences of the USSR
Miusskaya Sq.4, SU-125047, Moscow, USSR

Experiments on generating compilers by specializing specializers with respect to interpreters have shown that the compilers thus obtained have a natural structure only if the specializer does *variable splitting*. Variable splitting can result in a residual program using several variables to represent the values of a single variable of the original program. In the case of functional programming variable splitting is done by raising the arities of functions. The paper describes the structure and principles of operation of an arity raiser dealing with programs in a subset of pure Lisp.

Keywords: arity raiser, compiler generator, partial evaluation, retyping, specializer, variable splitting.

INTRODUCTION

Program specialization [Dixon 71] seems to be a promising and powerful technique that can lead to new program development methodology.

By *program specialization* we understand constructing, when given a "general-purpose" program and some restriction on its usage, a more efficient "specialized" residual program. Being optimized and simplified version of the original program, the residual program, however, must be equivalent to the original one when used according to the restriction. By *specializer* we understand a system that, given a program and a restriction, will produce a specialized version of the original program.

Program specialization can be achieved by making use of different techniques, such as *driving* [Turchin 72], *fold-unfold method* [Burstall 77], *partial evaluation* [Futamura 71], [Beckman 76], *mixed computation* [Ershov 78], [Bulyonkov 84], the *analysis of computational configurations* [Turchin 79], [Turchin 86], *variable splitting* [Sestoft 86], and *arity raising* [Romanenko 88].

The above techniques deal, for the most part, with two problems: *control restructuring* and *data retyping* (i.e. changing representation of data).

As far as the control restructuring is concerned, various specialization techniques differ in the extent to which the program is reorganized.

In the case of *monovariant* specialization any control point in the original program gives rise to zero or one control point in the residual program.

In the case of *polyvariant* specialization a control point can give rise to more than one control point in the residual program.

In the case of *monogenetic* specialization any control point in the residual program is produced from a single control point of the original program.

In the case of *polygenetic* specialization a control point in the residual program may be produced from several control points of the original program.

As far as the data representation is concerned, various specialization techniques differ in the use they make of *retyping*.

Driving [Turchin 72] and the analysis of configurations [Turchin 79], [Turchin 86], which deal with functional programs, can be classified as polyvariant polygenetic methods with retyping.

Monovariant monogenetic techniques for imperative programs are studied in [Ershov 78]. Papers [Bulyonkov 84], [Barzdin 88] concern polyvariant monogenetic specialization techniques for imperative programs.

The *transformational approach* [Ershov 81], [Ostrovski 88] is believed to include, at least potentially, all conceivable techniques of program specialization, not excluding the polygenetic ones.

Of course, the more powerful techniques tend to be rather expensive, and it is difficult to make them completely automatic. Thus the choice of appropriate specialization techniques depends on the class of problems to be solved.

An interesting application of specializers is compiler generation. It was found by Y.Futamura [Futamura 71] that interpreters can be converted to compilers by specializing a specializer with respect to the interpreters. Several years later it was realized [Beckman 76] that a transformer of interpreters into compilers can be produced by specializing a specializer with respect to a specializer.

To put this approach into practice, we have to overcome the following difficulty. On the one hand, the specializer has to be sophisticated enough to achieve non-trivial specialization. On the other hand, to be specializable, the specializer can't afford to be too complicated.

The group under N.D.Jones at Copenhagen university was the first to overcome the above difficulty [Jones 85], [Sestoft 86], [Sestoft 88]. Since experiments had shown the monovariant specialization to be unsatisfactory for this application, the specializer had to do the polyvariant specialization. Again, the monogenetic specialization proved to be adequate for the purpose (despite there being a lot of problems that have to be dealt with by polygenetic specialization [Turchin 82], [Wadler 88]).

The usefulness of retyping proved to be more problematic. It was found that retyping can be dispensed with at the cost of the residual programs having rather unnatural structure. Suppose, for example, that an interpreter is to be specialized with respect to a program. Since the interpreter is supposed to accept an arbitrary input program, the number of variables in this program cannot be known in advance. Thus the variable's values are likely to be represented in the interpreter as a single value assigned to one of the interpreter's variables. If the specializer is unable to split this variable, the residual program will use a single variable to represent all the values. A reasonable residual program, however, would keep each value in a separate variable [Sestoft 86].

To rectify the drawback, the author suggested that the Copenhagen specializer should be supplemented with an additional phase, whose purpose would be to do variable splitting [Romanenko 88]. In the case of a functional language, variable splitting reduces to increasing the number of functions' parameters, for which reason this additional phase was given the name *arity raiser*. As pointed out by T.Mogensen arity raising is just a special case of retyping, thus any arity raiser is a retyper.

The arity raiser was found to improve the structure of residual programs without making the specializer excessively slow and intricate.

The alternative to the arity raiser is to split variables *on-line*, i.e. at the time the residual program is being generated [Turchin 86], [Mogensen 88]. This approach, however, can result in a mammoth, sluggish specializer.

A short description of the ideas behind the arity raiser can be found in [Romanenko 88]. The present paper gives a detailed account of the structure and principles of operation of an arity raiser dealing with programs in a subset of pure Lisp.

1. THE LANGUAGE MIXWELL

In the following we consider programs written in the language Mixwell, which is a small subset of pure Lisp and was used as the subject language in the Copenhagen specializer MIX [Sestoft 86]. Here is Mixwell's abstract syntax.

pgm	∈	Program	programs
fd	∈	FnDef	function definitions
exp, e	∈	Exp	expressions
f	∈	FName	function names
x	∈	VName	variable names
$\mathcal{A}$	∈	Atom	Lisp atoms
$\mathcal{E}$	∈	SExp	Lisp S-expressions

pgm ::= fd_1; ... fd_n;

fd ::= $f(x_1, \ldots, x_m)$ = exp

exp ::= x

| **quote** $\mathcal{E}$

| **if** exp_0 **then** exp_1 **else** exp_2

| **call** $f(exp_1, \ldots, exp_m)$

| car(exp) | cdr(exp) | $cons(exp_1, exp_2)$

| atom(exp) | $equal(exp_1, exp_2)$

$\mathcal{E}$ ::= $\mathcal{A}$ | $(\mathcal{E}_1 \ . \ \mathcal{E}_2)$

A Mixwell program is a list of function definitions, the first function being the goal function. The goal function is to be called first, and inputs to the program are through the parameters of this function.

The body of a function is an expression, which is constructed from variables appearing in the function's formal parameter list, from constants **quote** and operators *car*, *cdr*, *cons*, *atom* and *equal* (as in

Lisp), conditionals **if** and defined function calls **call**.

The only data type is well-founded (i.e. non-circular) S-expressions as known from Lisp.

All primitive and defined functions, except the conditional **if**, are strict in all positions. All parameters are called by value.

We use some "sintactic sugar". The keyword **call** is omitted in cases where the name of the function being called is different from the names of the primitive functions. **quote** $\mathcal{E}$ can be written as '$\mathcal{E}$, cons(exp_1,exp_2) as exp_1 :: exp_2, equal(exp_1,exp_2) as exp_1 = exp_2. Constants ($\mathcal{E}_1$. ($\mathcal{E}_2$ . ... ($\mathcal{E}_n$. nil) ...)) can be written as ($\mathcal{E}_1$ $\mathcal{E}_2$ ... $\mathcal{E}_n$).

2. SPLITTING A FORMAL PARAMETER

Suppose the definition of function f in a program has the form

$$f(\ldots,x_k,\ldots) = exp$$

Then the following transformation will be referred to as *the splitting of the function's k-th parameter*.

Let x′ and x″ be two variables different from all formal parameters of the function f. Then the splitting of x into x′ and x″ can be done in two steps.

At the first step, the original definition of f is replaced with

$$f(\ldots,x',x'',\ldots) = exp[x_k \rightarrow cons(x',x'')]$$

where $exp[x_k \rightarrow cons(x',x'')]$ denotes the expression obtained from exp by replacing all occurrences of x_k with cons(x′,x″).

At the second step, all calls of the function f in all function definitions are transformed, each call of the form **call** f(...,e_k,...) being replaced with **call** f(..., car(e_k), cdr(e_k),...).

Thus, the original variable x_k is replaced by two new variables x′ and x″ containing enough information for the value of x_k, if needed, to be reconstructed. To put it more exactly, the value of x_k can be obtained by evaluating the expression cons(x′,x″).

The fact that the formal parameter x of the function f is to be split into two variables x′ and x″ will, for the brevity's sake, be written as f(x ⟶ x' :: x").

Example. Consider the program

```
f(x) = g(x :: x);  g(u) = cdr(u);
```

Let us perform the splitting g(u ⟶ u1 :: u2). After transforming the definition of g, we get

```
f(x) = g(x :: x);  g(u1,u2) = cdr(u1 :: u2);
```

Then we split the argument in the calls of g and get

```
f(x) = g(car(x :: x),cdr(x :: x));
g(u1,u2) = cdr(u1 :: u2);
```

This program can be locally optimized, which results in

```
f(x) = g(x,x);  g(u1,u2) = u2;
```

Now we see that variable splitting is capable of producing parameters whose values are certain not to be needed. Such parameters can be recognized by a kind of backward analysis [Hughes 88] and eliminated. In the above program we can remove the parameter u1 of the function g, which gives the program

```
f(x) = g(x);  g(u2) = u2;
```

Thus, the principal use of variable splitting consists in paving the way for other transformations such as local optimization and elimination of unneeded parameters, the latter being, in a sence, a kind of "garbage collection at compile time".

3. CONDITIONS OF THE VARIABLE SPLITTING CORRECTNESS

The program transformation described above can be incorrect. For example, after performing the splitting g(u ⟶ u1 :: u2) in the program

```
f(x) = g('a); g(u) = u;
```

we get

```
f(x) = g(car('a),cdr('a)); g(u1,u2) = u1 :: u2;
```

It is evident that the transformed program is not equivalent to the original one, because the original program terminates, with the result being the atom 'a, whereas the transformed program fails to apply *car* or *cdr* to the atom 'a and terminates abnormally. Thus we come to the conclusion:

> Before splitting a parameter, we must make sure that, when the program is run, it is impossible for the parameter's value to be an atom!

Hence, to split a variable, we need to have a description of the structure of its values. Such descriptions will be referred to as *types* of variables.

4. ANALYSIS OF RUN TIME TYPES

To describe the structure of values to be taken by a variable, we use the following set of types.

t	$\in$ Type	types
$\mathcal{A}$	$\in$ Atom	Lisp atoms

t	::=	*any*
	|	*atom*($\mathcal{A}$)
	|	*cons*(t_1, t_2)
	|	$\perp$

We assume the set of types to be equipped with reflexive partial ordering $\leq$ recursively defined by the following rules:

(i) $t \leq any$ for all types t.

(ii) $\perp \leq t$ for all types t.

(iii) $cons(t'_1, t'_2) \leq cons(t''_1, t''_2)$ if $t'_1 \leq t''_1$ and $t'_2 \leq t''_2$.

If $t' \leq t''$ and $t' \neq t''$, the type t″ is said to be *more general* than the type t'.

The set of types is a lattice, as for all types $t', t'' \in Type$ there exist their least upper bound $t' \sqcup t''$ and their greatest lower bound $t' \sqcap t''$. Each set of types $T \in \mathbb{P}(Type)$ has its least upper bound $\sqcup T$. Thus the set of types is a pointed continuous partial ordering (CPO) with the bottom $\bot$ [Schmidt 86]. It can be easily seen that the set of types has no chains of infinite height. In addition, each *finite* $T \in \mathbb{P}(Type)$ has its greatest lower bound $\sqcap T$.

A type represents a set of S-expressions. More specifically, let us define an "abstraction" function Abs mapping sets of S-expressions into types. Abs is defined in terms of an auxiliary function Abs' mapping S-expressions into types.

$Abs \in \mathbb{P}(SExp) \longrightarrow Type$

$Abs' \in SExp \longrightarrow Type$

$Abs[E] = \sqcup\{Abs'[\mathcal{E}] \mid \mathcal{E} \in E\}$

$Abs'[\mathcal{A}] = atom(\mathcal{A})$

$Abs'[(\mathcal{E}' . \mathcal{E}'')] = cons(\ Abs'[\mathcal{E}'],\ Abs'[\mathcal{E}''] \ )$

Let us define a "concretization" function Co reconstructing the set of S-expressions from a type:

$Co \in Type \longrightarrow \mathbb{P}(SExp)$

$Co[any] = SExp$

$Co[atom(\mathcal{A})] = \{\mathcal{A}\}$

$Co[cons(t',t'')] = \{(\mathcal{E}' . \mathcal{E}'') \mid \mathcal{E}' \in Co[t'] \text{ and } \mathcal{E}'' \in Co[t'']\}$

$Co[\bot] = \{\}$

The following relations hold:

$$Abs[Co[t]] = t \qquad \text{and} \quad E \subseteq Co[Abs[E]]$$

Now let x be a variable in a program. The problem is to find a type t such that $\mathcal{E} \in Co[t]$ for all $\mathcal{E}$ that can be taken as value by x when the program is run. It can be done by *abstract interpretation* [Jones 86] of the program, which amounts to performing the program's computations using abstract values in place of the actual ones.

Suppose we have a program defining functions $f_1, \ldots, f_h$. Let $F = \{f_1, \ldots, f_h\}$, and, for each $f \in F$, $x_{f,j}$ be its j-th parameter, $a(f)$ be its arity, and $body_f$ be its body, so that the definition of f has the form:

$$f(x_{f,1}, \ldots, x_{f,a(f)}) = body_f$$

Let

$\theta \quad \in Env = VName \longrightarrow Type$

be an *environment* assigning a type to each parameter of a function. Let

$\alpha \quad \in ArgDescr = F \longrightarrow Env$

be an *argument type description* assigning types to each function's parameters. Let

$\rho \quad \in ResDescr = F \longrightarrow Type$

be a *result type description* assigning a type to each function's result.

All the sets above are equipped with reflexive partial orderings as follows:

Env:	$\theta' \le \theta'' \Leftrightarrow \forall x \in VName\ \theta'(x) \le \theta''(x)$
ArgDescr:	$\alpha' \le \alpha'' \Leftrightarrow \forall f \in F\ \alpha'(f) \le \alpha''(f)$
ResDescr:	$\rho' \le \rho'' \Leftrightarrow \forall f \in F\ \rho'(f) \le \rho''(f)$

We define two functions R and A to do the abstract interpretation using these ordered sets.

The function R, given an expression exp, an environment θ, and a result type description ρ, computes the type of an expression's result.

$R \quad \in Exp \longrightarrow Env \longrightarrow ResDescr \longrightarrow Type$

$R[x]\ \theta\ \rho = \theta(x)$

$R[\textbf{quote}\ \mathcal{E}]\ \theta\ \rho = Abs'[\mathcal{E}]$

$R[\textbf{if}\ exp\ \textbf{then}\ exp'\ \textbf{else}\ exp'']\ \theta\ \rho =$

$\quad R[exp']\ \theta\ \rho \sqcup R[exp'']\ \theta\ \rho$

$R[\textbf{call}\ f(exp_1, \ldots, exp_m)]\ \theta\ \rho = \rho(f)$

$$R[\mathrm{car(exp)}]\ \theta\ \rho = \begin{cases} \mathit{any} & \text{if } R[\mathrm{exp}]\ \theta\ \rho = \mathit{any}, \\ t' & \text{if } R[\mathrm{exp}]\ \theta\ \rho = \mathit{cons}(t', t''), \\ \bot & \text{otherwise.} \end{cases}$$

$$R[\mathrm{cdr(exp)}]\ \theta\ \rho = \begin{cases} \mathit{any} & \text{if } R[\mathrm{exp}]\ \theta\ \rho = \mathit{any}, \\ t'' & \text{if } R[\mathrm{exp}]\ \theta\ \rho = \mathit{cons}(t', t''), \\ \bot & \text{otherwise.} \end{cases}$$

$R[\mathrm{cons(exp', exp'')}]\ \theta\ \rho =$

$\qquad \mathit{cons}(\ R[\mathrm{exp'}]\ \theta\ \rho,\ R[\mathrm{exp''}]\ \theta\ \rho\ )$

$R[\mathrm{atom(exp)}]\ \theta\ \rho = \mathit{any}$

$R[\mathrm{equal(exp', exp'')}]\ \theta\ \rho = \mathit{any}$

The function A, given an expression exp, an environment θ, an argument type description α, and a result type description ρ, computes a new approximation to the final description of each function's parameter values.

$A \quad \in \mathrm{Exp} \longrightarrow \mathrm{Env} \longrightarrow \mathrm{ArgDescr} \longrightarrow \mathrm{ResDescr} \longrightarrow \mathrm{ArgDescr}$

$A[x]\ \theta\ \alpha\ \rho = \alpha$

$A[\textbf{quote}\ \mathcal{E}]\ \theta\ \alpha\ \rho = \alpha$

$A[\textbf{if}\ \mathrm{exp}\ \textbf{then}\ \mathrm{exp'}\ \textbf{else}\ \mathrm{exp''}]\ \theta\ \alpha\ \rho =$

$\qquad A[\mathrm{exp}]\ \theta\ \alpha\ \rho \sqcup A[\mathrm{exp'}]\ \theta\ \alpha\ \rho \sqcup A[\mathrm{exp''}]\ \theta\ \alpha\ \rho$

$A[\textbf{call}\ f(\mathrm{exp}_1, \ldots, \mathrm{exp}_m)]\ \theta\ \alpha\ \rho =$

$\qquad \alpha_{new}[f \mapsto \alpha_{new}(f) \sqcup \theta_{new}]$, where

$\qquad\qquad \alpha_{new} = \sqcup\ \{A[\mathrm{exp}_j]\ \theta\ \alpha\ \rho\}_{j=1,\ldots,m}$ and

$\qquad\qquad \theta_{new} = [x_{f,j} \mapsto R[\mathrm{exp}_j]\ \theta\ \rho]_{j=1,\ldots,m}$

$A[\mathrm{car(exp)}]\ \theta\ \alpha\ \rho = A[\mathrm{exp}]\ \theta\ \alpha\ \rho$

$A[\mathrm{cdr(exp)}]\ \theta\ \alpha\ \rho = A[\mathrm{exp}]\ \theta\ \alpha\ \rho$

$A[\mathrm{cons(exp', exp'')}]\ \theta\ \alpha\ \rho = A[\mathrm{exp'}]\ \theta\ \alpha\ \rho \sqcup A[\mathrm{exp''}]\ \theta\ \alpha\ \rho$

$A[\mathrm{atom(exp)}]\ \theta\ \alpha\ \rho = A[\mathrm{exp}]\ \theta\ \alpha\ \rho$

$A[\mathrm{equal(exp', exp'')}]\ \theta\ \alpha\ \rho = A[\mathrm{exp'}]\ \theta\ \alpha\ \rho \sqcup A[\mathrm{exp''}]\ \theta\ \alpha\ \rho$

We want a final argument type description α that is consistent and as low as possible. This description can be determined by finding the least fixed point for the following system of simultaneous equations and relations:

$$\alpha = \sqcup \{A[body_f]\ \alpha(f)\ \alpha\ \rho\}_{f\in F}, \qquad \alpha \geq \alpha_0$$

$$\rho = [f \mapsto R[body_f]\ \alpha(f)\ \rho]_{f\in F} \qquad \rho \geq \rho_0$$

where α_0 and ρ_0 are defined as follows

$$\alpha_0 = [f_1 \mapsto [x_{f_1,j} \mapsto any]_{j=1,\dots,a(f_1)}] \sqcup$$

$$[f \mapsto [x_{f,j} \mapsto \perp]_{j=1,\dots,a(f)}]_{f\in F}$$

$$\rho_0 = [f \mapsto \perp]_{f\in F}$$

The description α_0 assigns the type *any* to the parameters of the goal function f_1, since an input parameter may be given as value an arbitrary S-expression. All other parameters, on the contrary, are assigned the type $\perp$, there being no *a priory* information about their possible values.

The description ρ_0 assigns the type $\perp$ to the results of all functions.

The least fixed point for the above system does exist because for any given program the ordered sets involved have no chains of infinite height, and the functions A and R are monotonic.

To do variable splitting, we need only the argument type description α, the result type description ρ being used only during the analysis of types.

The type analysis above can, in a sense, be regarded as a monovariant, monogenetic version of the "configuration analysis" as used in the Supercompiler [Turchin 89], [Turchin 86].

5. USING TYPE INFORMATION FOR VARIABLE SPLITTING

The variable splitting transformation as described above splits only one of a function's parameters. However, the information provided by an argument type description is sufficient for all function's parameters to be split at once.

Suppose a parameter x has the type t. If t contains some occurrences of $\perp$, $Co[t] = \{\}$ holds, which implies that no S-expression can be taken as value by x, and therefore the function to which the parameter belongs never will be called. In this case, all calls of the function can be replaced with any construct that forces the program to abnormally terminate (for example, with car(**quote** nil)), and thereafter, the definition of the function can be eliminated from the program.

For this reason we assume, henceforth, the type t of any variable to be non-empty, i.e. to satisfy the condition Co[t] ≠ {}.

In the general case a type t assigned to a variable x may contain some occurrences of the type *any*, which are referred to as "gaps".

It is obvious that all values of the variable x can be different only at places corresponding to the gaps, and must be congruent at all other places. Therefore, if the type t contains *m* gaps, any S-expression $\mathcal{E}\in$Co[t] is completely determined by its parts corresponding to the gaps in the type t. This enables the variable x to be *retyped* by replacing it with *m* new variables, which are to be assigned the parts of the variable's values corresponding to the gaps.

We use the following notation. A finite list of elements a_1, ..., a_m is written as [a_1, ..., a_m], an empty list as []. The length of a list A is written as len(A). The concatenation of two lists A = [a_1, ..., a_m] and B = [b_1, ..., b_n] equal to [a_1, ..., a_m, b_1, ..., b_n] is denoted by A^B.

Given a type t, a variable x, and a list of new variables [x_1, ..., x_m], it is easy to construct an expression synthesizing the value of the original variable x from values of the new variables. With this aim in view, let us define a few functions.

The function CountGaps, given a type, produces the natural number equal to the number of gaps in the type.

CountGaps ∈ Type ⟶ ℕ

CountGaps[*any*] = 1

CountGaps[*atom*($\mathcal{A}$)] = 0

CountGaps[*cons*(t′,t″)] = CountGaps[t′] + CountGaps[t″]

The function ExpandVar, given a type and a list of new variables, constructs an expression synthesizing the original value from the values of the new variables. The length of the variable list must be equal to the number of gaps in the type.

ExpandVar $\in$ Type $\longrightarrow$ VName* $\longrightarrow$ Exp

ExpandVar[*any*] [x] = x

ExpandVar[*atom*($\mathcal{A}$)] [] = **quote** $\mathcal{A}$

ExpandVar[*cons*(t′,t″)] X = cons(ExpandVar[t′] X′ , ExpandVar[t″] X″)

where X′^X″ = X, len(X′) = CountGaps[t′]

and len(X″) = CountGaps[t″].

Let us consider an argument expression exp appearing in a function call. Let x be the corresponding formal parameter, t be the type assigned to x, and x_1, ..., x_m be *m* new parameters into which the parameter x is to be split. Then the expression exp is to be split into *m* new expressions exp_1, ..., exp_m such that each exp_j will produce the value to be assigned to the new parameter x_j. The function SplitArg can easily be defined which, given a type, an expression, and a list of new variables, produces a list of expressions resulting from splitting the original expression.

SplitArg $\in$ Type $\times$ Exp $\longrightarrow$ Exp*

SplitArg[*any*, exp] = [exp]

SplitArg[*atom*($\mathcal{A}$), exp] = []

SplitArg[*cons*(t′,t″), exp] =

[SplitArg[t′,car(exp)]] ^ [SplitArg[t″,cdr(exp)]]

Now we are able to describe the splitting of variables throughout the program. This can be done in two steps.

At the first step, the splitting is performed of all formal parameters of the functions. Each function definition

$$f(x_{f,1},\ldots,x_{f,a(f)}) = body_f$$

is treated as follows.

First, each formal parameter is split. Let a parameter x have the type t. Then a list of new variables X = [x_1, ..., x_m] is created consisting of *m* = CountGaps[t] new parameters, the new parameters being different from all other formal parameters of the function f. After that, x is replaced with the sequence of *m* new parameters x_1, ..., x_m.

Thereafter, the transformation is performed of the function's body

$body_f$, which results in all occurrences of the formal parameters being replaced with new expressions. To put it more exactly, all occurrences of a parameter x are replaced with the expression ExpandVar[t] X, where t is the type of x, and X = [x_1, ..., x_m] is the list of the variables that x has been split into.

At the second step, the splitting is performed of the argument expressions by replacing each argument expression exp with the expression sequence exp_1, ..., exp_m, where [exp_1, ..., exp_m] = SplitArg[t,exp], and t is the type assigned to the corresponding formal parameter.

An actual implementation of the above transformation can do the replacing of parameters with new expressions and the splitting of argument expressions simultaneously.

6.CODE DUPLICATION RISK

Example. Consider the program

```
f(z) = swap(unzip(z,'nil,'nil));
unzip(u,x,y) = if u='nil then x :: y else
     unzip(cdr(u),car(car(u)) :: x, cdr(car(u)) :: y);
swap(v) = cdr(v) :: car(v);
```

It is evident that any result produced by the function unzip is of the type *cons*(*any*,*any*), hence this type can be assigned to the parameter v of the function swap. Thus we are allowed to perform the splitting swap(v ⟶ v1 :: v2), which gives the program

```
f(z) = swap(car(unzip(z,'nil,'nil)),
               cdr(unzip(z,'nil,'nil)));
unzip(u,x,y) = if u='nil then x :: y else
     unzip(cdr(u),car(car(u)) :: x, cdr(car(u)) :: y);
swap(v1,v2) = v2 :: v1;
```

We see that the transformation has given rise to two copies of the expression unzip(z,'nil,'nil). This is bad for two reasons. First, duplicating expressions can result in huge programs being produced. Second, code duplication can lead to repeated evaluation of expressions.

Both of the problems arise in the above example.

The risk of code duplication and repeated evaluation can be avoided by the following principle of "selector non-introduction":

> All selectors produced by variable splitting must be eliminable by means of local optimization.

What is the drawback of the type analysis described above? The point is that this analysis tells us whether a selector in the program is certain to be applicable at run time, whereas we need to know whether the selector can be applied *symbolically* at the time the program is being optimized.

To put it another way, when an argument expression exp is to be split into the two expressions car(exp) and cdr(exp), the expression exp should have the structure permitting the selectors car and cdr to be eliminated by local optimization.

The feasibility of the simbolic application of a selector to the expression exp, obviously, depends upon the structure of the expression itself, rather than on the structure of the result to be produced by exp at run time.

Let us consider a few different cases.

If exp has the form **quote** ($\mathcal{E}'$. $\mathcal{E}''$), the symbolic application is feasible, car(exp) being reducible to **quote** $\mathcal{E}'$, and cdr(exp) being reducible to **quote** $\mathcal{E}''$.

If exp has the form exp′ :: exp″, the symbolic application is feasible, car(exp) being reducible to exp′, and cdr(exp) being reducible to exp″.

On the other hand, if exp has the form **if** exp_0 **then** exp′ **else** exp″ or **call** f(exp_1,...,exp_m), it is impossible to make the symbolic application without code duplication.

If exp is a variable x, the symbolic application may seem to be unfeasible, because car(exp) is car(x), and cdr(exp) is cdr(x). Thus, the selectors cannot be eliminated. Consider, however, the following example.

Example. Suppose we have the program

```
f(x) = g(x :: x);  g(u) = h(u);  h(v) = cdr(v);
```

It is obvious that the parameter u of the function g can be split, since the argument expression has the form exp′ :: exp″. On the other hand, the argument expression in the call of the function h is a variable, and, for this reason, splitting the parameter v seems to be unfeasible. Nevertheless, after g(u ⟶ u1 :: u2), we get the program

```
f(x) = g(x,x);  g(u1,u2) = h(u1 :: u2);
h(v) = cdr(v);
```

We see, now, that splitting the parameter u results in the argument expression u of the function h being replaced with the expression u1 :: u2, which is easy to split! After performing h(v ⟶ v1 :: v2), we get

```
f(x) = g(x,x);  g(u1,u2) = h(u1,u2);
h(v1,v2) = v2;
```

Thus, if an expression to be split consists of a single variable, then, instead of analyzing the original expression, we have to analyze the new expression by which the original one will be replaced because of the parameters being split throughout the program.

7. ANALYSIS OF OPTIMIZATION TIME TYPES

As can be seen from the above, we need to know the structure of *symbolic* values assigned to variables at the time the program is being optimized, rather than the structure of ordinary values assigned to variables at the time the program is run. Thus, what we are really interested in are the optimization time types, rather than the run time types.

To find them, we can use the same set of types as has been used for analyzing the run time types.

As pointed out previously, no call of a defined function can be split without being duplicated. Thus, the results of defined functions have to be assigned the type *any*. For this reason the result type description can be dispenced with, which enables the analysis of types to be simplified, the only description needed being the argument type description. Hence, the above functions R and A have to be redefined.

The function R, given an expression exp and an environment θ, computes the type of an expression's result.

R $\in$ Exp $\longrightarrow$ Env $\longrightarrow$ Type

R[x] θ = θ(x)

R[**quote** $\mathcal{E}$] θ = Abs'[$\mathcal{E}$]

R[**if** exp **then** exp' **else** exp"] θ = *any*

R[**call** f(exp_1,...,exp_m)] θ = *any*

$$R[car(exp)]\ \theta = \begin{cases} any & \text{if } R[exp]\ \theta = any, \\ t' & \text{if } R[exp]\ \theta = cons(t',t''), \\ \bot & \text{otherwise.} \end{cases}$$

$$R[cdr(exp)]\ \theta = \begin{cases} any & \text{if } R[exp]\ \theta = any, \\ t'' & \text{if } R[exp]\ \theta = cons(t',t''), \\ \bot & \text{otherwise.} \end{cases}$$

R[cons(exp',exp")] θ =

cons(R[exp'] θ, R[exp"] θ)

R[atom(exp)] θ = *any*

R[equal(exp',exp")] θ = *any*

The function A, given an expression exp, an environment θ, and an argument type description α, computes a new approximation to the final description of each function's parameter types.

A $\in$ Exp $\longrightarrow$ Env $\longrightarrow$ ArgDescr $\longrightarrow$ ArgDescr

A[x] θ α = α

A[**quote** $\mathcal{E}$] θ α = α

A[**if** exp **then** exp' **else** exp"] θ α =

A[exp] θ α $\sqcup$ A[exp'] θ α $\sqcup$ A[exp"] θ α

A[**call** f(exp_1,...,exp_m)] θ α =

$\alpha_{new}[f \mapsto \alpha_{new}(f) \sqcup \theta_{new}]$, where

$\alpha_{new} = \sqcup \{A[exp_j]\ \theta\ \alpha\}_{j=1,\ldots,m}$ and

$$\theta_{new} = [x_{f,j} \mapsto R[exp_j]\ \theta]_{j=1,\dots,m}$$

$A[car(exp)]\ \theta\ \alpha = A[exp]\ \theta\ \alpha$

$A[cdr(exp)]\ \theta\ \alpha = A[exp]\ \theta\ \alpha$

$A[cons(exp', exp'')]\ \theta\ \alpha = A[exp']\ \theta\ \alpha \sqcup A[exp'']\ \theta\ \alpha$

$A[atom(exp)]\ \theta\ \alpha = A[exp]\ \theta\ \alpha$

$A[equal(exp', exp'')]\ \theta\ \alpha = A[exp']\ \theta\ \alpha \sqcup A[exp'']\ \theta\ \alpha$

We want a final argument type descripton α that is consistent and as low as possible. This must be the least fixed point for the following system of simultaneous equations and relations:

$$\alpha = \sqcup\ \{A[body_f]\ \alpha(f)\ \alpha\}_{f\in F}, \qquad \alpha\geq\alpha_0$$

where α_0 is defined as follows

$$\alpha_0 = [f_1 \mapsto [x_{f_1,j} \mapsto any]_{j=1,\dots,a(f_1)}] \sqcup$$
$$[f \mapsto [x_{f,j} \mapsto \bot]_{j=1,\dots,a(f)}]_{f\in F}$$

The description α_0 assigns the type *any* to the parameters of the goal function f_1, to prevent these parameters from being split. All other parameters, on the contrary, are assigned the type $\bot$, there being no *a priori* information about their structure.

The least fixed point for the system above does exist because for any given program the ordered sets involved have no chains of infinite height, and the functions A and R are monotonic.

8. USEFULNESS OF VARIABLE SPLITTING

The fact that the parameters of a function f have been assigned the types $t_1, \dots, t_m$, for brevity's sake, will be written as $f(t_1,\dots,t_m)$.

Let us consider the following example.

<u>Example</u>.

```
f(x) = rev(x, 'a :: 'nil);
rev(u,v) = if u = 'nil then v else
               rev(cdr(u), car(u) :: v);
```

The analysis of types tells us that f(*any*), rev(*any*,*cons*(*any*,*any*)). After rev(v ⟶ v1 :: v2), we get the program

```
f(x) = rev(x, 'a, 'nil);
rev(u,v1,v2) = if u = 'nil then v1 :: v2 else
                   rev(cdr(u), car(u), v1 :: v2);
```

We see that the program obtained is by no means superior to the original one, because no selector has been eliminated owing to variable splitting.

Thus we see that the parameter splitting based exclusively on the information obtained by examining the structure of argument expressions, may well result in the "arity overraising", i.e. increasing the number of parameters without reducing the number of selectors in the program. The types as produced by the above analysis, describing as they do the feasibility of splitting parameters, however, provide no information on the usefulness of this splitting. The arity overraising, nevertheless, can be avoided by "adjusting" the above types in the following way.

Suppose, for example, the type t has been assigned to a parameter x. Then the splitting of the parameter can be restricted by replacing some parts of t having the form *cons*(t_1,t_2) with *any*. This results in the type t being *generalized*, i.e. changed to some other type t′ such that t ≤ t′, the depth of splitting being the less the greater the type t′. Thus, for instance, the splitting x ⟶ x1 :: (x2 :: x3) corresponds to the type *cons*(*any*, *cons*(*any*,*any*)), the splitting x ⟶ x1 :: x2 to the type *cons*(*any*,*any*), and no splitting to the type *any*.

Thus we are facing the *type generalization problem:* given a *cons* in a type, we have to decide whether this *cons* should be retained or generalized. This decision will be made on the basis of the following *selector elimination principle:*

> A *cons* should be retained only if this causes a selector in the program to disappear.

Being formalized as it is, the selector elimination principle gives only an approximate description of the intuitive ideas the humans have about what does it means for a program to have a beautiful and natural structure. Nevertheless, experience has shown this principle to be likely to produce reasonable results, without any danger of the program being spoilt.

9. BACKWARD ANALYSIS

Let us consider the definition of function f

$$f(\ldots,x_k,\ldots) = exp$$

The *k*-th parameter of the function may appear at different places in the function's body exp. Is it any use splitting x_k? To answer this question, we have to consider all occurrences of x_k in exp and to take into account their *contexts* in exp. To take an example, if exp contains the subexpression cdr(x_k), it makes sense to perform the splitting $x_k \rightarrow x' :: x''$, since this will cause cdr(x_k) to be replaced with cdr(x' :: x''), the latter being reducible to x''.

Example.

f(x) = g(x :: x); g(u) = u;

In this case the selector elimination principle tells us that it is no use performing the splitting g(u ⟶ u1 :: u2).

Example.

f(x) = g(x :: x); g(u) = cdr(u);

In this case the selector elimination principle tells us that the splitting g(u ⟶ u1 :: u2) is worth performing, since it will cause the selector cdr to disappear. And, in fact, after the splitting we get the program

f(x) = g(x,x); g(u1,u2) = u2;

Thus we see that the natural way of getting information about the usefulness of splitting is to make use of some kind of backward analysis [Hughes 88].

10. ACCESS PATHS AND CONTEXTS

Let exp be an expression appearing in a larger expression. We want to consider all attempts by the surrounding expression at accessing the components of exp. For example, if exp is a part of the expression

car(cdr(cdr(exp)))

then there is an attempt at accessing exp by applying selectors in the following order: cdr, cdr, car. The component to be accessed can be unambiguously identified by a sequence of selectors. This justifies the following definition.

Definition. An *access path* is a finite list (which may be empty) of selector names car and cdr.

The set of all access paths will be denoted by Path. Thus Path = $\{car, cdr\}^*$.

In some cases the surrounding expression tries to access several components of the expression under consideration. For this reason we have to describe the context by a set of paths, rather than by a single path.

Definition. A set of access paths $\Pi \in \mathbb{P}$(Path) is an *access context*, if it satisfies the following requirements.

(i) $[\,] \in \Pi$

(ii) If π^[car] $\in \Pi$ or π^[cdr] $\in \Pi$, then $\pi \in \Pi$.

(i) means that an attempt at accessing the expression as a whole must be included into the context. This requirement is useful for technical reasons. (ii) formalizes the obvious fact that a subcomponent can be accessed only by accessing the components in which the subcomponent is included.

The set of all contexts is denoted by Context.

Now consider function f with the definition

$f(\ldots, x_k, \ldots)$ = exp

Suppose that exp contains *m* occurrences of the parameter x_k in the contexts Π_1, $\Pi_2, \ldots, \Pi_m$. What should be the total context for all occurrences of x_k? It is clear that finding all attempts at accessing the parameter x_k amounts to finding all attempts at accessing its occurrences, thus $\Pi_1 \cup \Pi_2 \cup \ldots \cup \Pi_m$ should be considered to be the total context of the parameter x_k.

11. USING CONTEXTS FOR TYPE GENERALIZATION

Let a parameter have the type t and the context Π. Then the function GenType can be easily defined which generalizes t in accordance with Π by replacing all $cons(t_1, t_2)$ unaccessed by Π with *any*.

GenType : Type ⟶ Context ⟶ Type

GenType[t] Π = $\sqcap$ {GenType'[t]π | π∈Π}

GenType' : Type ⟶ Path ⟶ Type

GenType'[*any*]π = *any*
GenType'[*atom*($\mathcal{A}$)]π = *atom*($\mathcal{A}$)
GenType'[*cons*(t',t")]([]) = *any*
GenType'[*cons*(t',t")]([car]^π) = *cons*(GenType'[t']π, *any*)
GenType'[*cons*(t',t")]([cdr]^π) = *cons*(*any*, GenType'[t"]π)
GenType'[⊥]π = ⊥

It should be noted that for all t∈Type and π∈Π the relation t ≤ GenType'[t]π holds, therefore the set {GenType'[t]π | π∈Π} is finite, in spite of the fact that Π may well be infinite. Consequently, the greatest lower bound of this set does exist.

12. LATENT SELECTORS

The above considerations might have produced the expression that the context of a parameter can be determined by examining only the definition of the function concerned, without the program being globally analyzed. This is not really the case, however.

Example.

```
f(x) = g(x :: 'a); g(u) = h(u); h(v) = cdr(v);
```

The type analysis tells us that f(*any*), g(*cons*(*any*, *atom*(a))), h(*cons*(*any*, *atom*(a))). The variable v has the context {[], [cdr]}. But what is the context of the variable u? At the first glance, it may appear to be {[]}, because there seems to be no selectors in the program attempting at accessing the variable u. Thus we, erroneously, come to the conclusion that the types should be generalized as follows: f(*any*), g(*any*), h(*cons*(*any*, *atom*(a))). The only acceptable splitting is therefore h(v ⟶ v1 :: v2). By performing it we get

```
f(x) = g(x :: 'a); g(u) = h(car(u),cdr(u));
h(v1,v2) = v2;
```

This result is far from being satisfactory, because there have appeared two new selectors car and cdr, not present in the original program. This makes us draw the conclusion that the parameter access analysis has to take into account not only the selectors explicitly appearing in the program, but also the *latent selectors* to be introduced by the splitting of parameters.

Thus, if e_k is an argument expression in the function call **call** f(...,e_k,...), it would be incorrect to take its context to be {[]}, because there should be taken into account all attempts at accessing e_k due to the splitting of e_k. This can be done in the following way.

Let the *k*-th formal parameter of the function f be assigned the type t, and the total context of all its occurrences be Π. Let t′ = GenType[t]Π. Then the generalized type t′ gives all information about the way in which e_k is to be split. The function TypeToContext can be easily defined which converts t′ into the context providing the information about all the attempts at accessing e_k due to the splitting of e_k in accordance with t′.

TypeToContext : Type ⟶ Context

TypeToContext[*any*] = {[]}
TypeToContext[*atom*($\mathcal{A}$)] = {[]}
TypeToContext[*cons*(t′,t″)] = {[]} ∪
    car*(TypeToContext[t′]) ∪ cdr*(TypeToContext[t″])
TypeToContext[⊥] = {[]}

where we use the notation

$$\text{car}^*\Pi = \{[\text{car}]\hat{}\pi \mid \pi\in\Pi\}$$
$$\text{cdr}^*\Pi = \{[\text{cdr}]\hat{}\pi \mid \pi\in\Pi\}.$$

Now we can determine the context of the expression e_k, assuming the k-th parameter to be assigned the type t, and the total context of all its occurrences to be Π. This context is equal to

$$\text{TypeToContext}[\ \text{GenType}[t]\Pi\ ]$$

13. SYSTEM OF EQUATIONS FOR FINDING CONTEXTS

For each function f with the definition

$$f(x_{f,1},\ldots,x_{f,m}) = \text{body}_f$$

let $t_{f,1}$, ..., $t_{f,m}$ stand for the types of its parameters, and $c_{f,1}$, ..., $c_{f,m}$ stand for the contexts of its parameters.

Let C x [exp] Π be the total context of all occurrences of the variable x in the expression exp, the expression exp itself being in the context Π.

We have the following set of equations

$$c_{f,j} = \text{TypeToContext}[\ \text{GenType}[t_{f,j}]\ (\ C\ x_j\ [\text{body}_f]\ \{[\,]\}\ )\ ]$$

where C x [exp] Π is defined as follows:

$$C \quad \in \text{VName} \longrightarrow \text{Exp} \longrightarrow \text{Context} \longrightarrow \text{Context}$$

$C\ x\ [x]\ \Pi = \Pi$

$C\ x\ [y]\ \Pi = \{[\,]\}$, where $x \neq y$.

C x [**quote** $\mathcal{E}$] Π = {[ ]}

C x [**if** exp **then** exp′ **else** exp″] Π =

C x [exp] {[ ]} ∪ C x [exp′] {[ ]} ∪ C x [exp″] {[ ]}

C x [**call** $f(\text{exp}_1,\ldots,\text{exp}_m)$] Π =

$$\cup\{\ C\ x\ [\text{exp}_j]\ c_{f,j}\ \}_{j=1,\ldots,m}$$

C x [car(exp)] Π = C x [exp] ({[ ]} ∪ car*Π)

C x [cdr(exp)] Π = C x [exp] ({[ ]} ∪ cdr*Π)

$C\ x\ [cons(exp',exp'')]\ \Pi =$

$$C\ x\ [exp']\ (\{[\,]\} \cup \Pi/car) \cup C\ x\ [exp'']\ (\{[\,]\} \cup \Pi/cdr)$$

$C\ x\ [atom(exp)]\ \Pi = C\ x\ [exp]\ \{[\,]\}$

$C\ x\ [equal(exp',exp'')]\ \Pi = C\ x\ [exp']\ \{[\,]\} \cup C\ x\ [exp'']\ \{[\,]\}$

where we use the notation

$$\Pi/car = \{\pi \mid [car]\hat{}\pi \in \Pi\}$$
$$\Pi/cdr = \{\pi \mid [cdr]\hat{}\pi \in \Pi\}$$

We assume the set of contexts to be equipped with natural partial ordering, $\Pi' \le \Pi''$ being equivalent to $\Pi' \subseteq \Pi''$. The functions TypeToContext, GenType, and C are monotonic with respect to contexts, therefore the minimal fixed point for the above system of equations does exist.

Moreover, since $c_{f,j} \subseteq$ TypeToContext$[t_{f,j}]$, there exist only a finite number of contexts that can be taken as value by $c_{f,j}$, hence the minimal fixed point can be found by a finite number of iterations.

The context analysis above resembles, in some respects, the "neighborhood analysis" as used in the Supercompiler [Turchin 86], [Turchin 88].

14. PRACTICAL IMPLEMENTATION OF THE CONTEXT ANALYSIS

Some programming tricks may prove to be useful for implementing the above backward analysis.

First, what we really use in splitting parameters are types generalized with respect to contexts, rather than contexts themselves. Thus, instead of computing $c_{f,j}$, we can compute the type $t'_{f,j}$ = GenType$[t_{f,j}]\ c_{f,j}$.

Second, since $t_{f,j} \le t'_{f,j}$, we can replace $t_{f,j}$ and $t'_{f,j}$ with a single *marked type* $mt_{f,j}$ having the syntax

mt	∈ MType		marked types
mt	::=	*any*	
	|	*atom*($\mathcal{A}$)	
	|	*cons*(t′,t″)	
	|	*cons*!(mt′,mt″)	
	|	⊥	

The feature of the marked type is that some *cons* are marked with the exclamation mark, which indicates that these *cons* belong both to the type $t'_{f,j}$ and to the type $t_{f,j}$. On the contrary, if a *cons* is not marked, it means that it belongs only to $t_{f,j}$, the corresponding place in $t'_{f,j}$ being *any*.

Thus, $t_{f,j}$ can be extracted from $mt_{f,j}$ by removing all exclamation marks, whereas $t'_{f,j}$ can be extracted from $mt_{f,j}$ by replacing all *cons*! with *cons*, and all *cons*(t',t") with *any*. If we need, however, the context $c_{f,j}$, rather than the type $t'_{f,j}$, the context $c_{f,j}$ can be extracted from $mt_{f,j}$ directly, without finding $t'_{f,j}$, by means of the function Restore!.

Restore! ∈ MType ⟶ Context

Restore![*any*] = {[]}

Restore![*atom*(𝒜)] = {[]}

Restore![*cons*(t',t")] = {[]}

Restore![*cons*!(mt',mt")] = {[]} ∪
    car*Restore![mt'] ∪ cdr*Restore![mt"]

Next improvement concerns the representation of contexts. Being sets of paths, contexts are difficult to deal with directly, but we can replace contexts with their representations having the syntax

crep ∈ ContextRep

crep ::= *car*(crep)
    | *cdr*(crep)
    | *mtype*(mt)

Given a context's representation, we can reconstruct the context by the function Restore.

Restore ∈ ContextRep

Restore[*car*(crep)] = {[]} ∪ car*Restore[crep]

Restore[*cdr*(crep)] = {[]} ∪ cdr*Restore[crep]

Restore[*mtype*(mt)] = Restore![mt]

As a matter of fact, all functions which the access path analysis involves can be easily modified so that they will deal with the representation of contexts, rather than with the contexts themselves.

15. GENERALIZATIONS

The first obvious generalization concerns splitting the results of functions. In the language Mixwell each function has *fixed arity*, which means that this function is to be given a fixed number of arguments (zero, one, or more). On the other hand, each function can produce one and only one result, for which reason we had to draw the conclusion that a defined function call **call** f(exp_1,...,exp_m) cannot be split and, therefore, has to be assigned the type *any*. Nevertheless, in addition to the concept of arity, we can introduce the concept of *coarity* by letting each function produce a fixed number of results (zero, one, or more). The implementation of multi-result functions can cause no problems: when a function having the arity m and the coarity n is to be called, we have to put m values into the parameter stack. Then the function takes from the stack the input values and pushes onto the stack n output values.

A function being able to produce several results allows the function's results to be split without splitting the definition of the function. This extention of the arity raiser has been implemented by Ruten Gurin.

Another possible extension is to make an arity raiser deal with data structures that are more complicated than Lisp S-expressions are. To take an example, in the case of the languages Refal [Turchin 79], [Turchin 86] and RL [Romanenko 88], the main data type is the set of *object expressions*. The difficulty is that, instead of the single constructor *cons*, we have two constructors: "enclosing an expression in parentheses" and "concatenating two expressions". The concatenation is an associative operation, and the last element of an expression can be accessed as well as its first element. A consequence is that two types t′ and t″ may happen not to have the least upper bound, in which case, during the type analysis, we have to satisfy ourselves with finding a type t such that $t' \le t$, $t'' \le t$, but t is not "too high". The arity raiser built into the "Moscow" specializer [Romanenko 88], uses a set of types with the above feature.

CONCLUSIONS

In order for the results produced by variable splitting to be reasonable, we need information obtained by two preliminary global analyses of the program. The first, forward, analysis tells us whether the splitting is feasible, whereas the second, backward, analysis tells us whether the splitting is useful.

The first, forward, analysis results in the parameters of the functions being assigned *types*, which describe the structure of the argument expressions in the function calls.

The second, backward, analysis results in the parameters of the functions being assigned *contexts*, which provide information about attempts at accessing the parameters.

Then the information obtained is used to perform variable splitting. The type information is used to avoid introducing new selectors into the program as well as code duplication, whereas the context information makes it possible to avoid useless variable splitting that does not cause some selectors in the program to be eliminated.

Introducing an arity raiser as a separate phase into a specializer enhances the structure of residual programs generated without affecting the other phases of the specializer. The structure of the specializer, thus, can be kept natural and understandable.

REFERENCES

[Barzdin 88]

G.Barzdin. Mixed Computation and Compiler Basis. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 15-26, North-Holland, 1988.

[Beckman 76]

L.Beckman, A.Haraldson, O.Oskarsson, E.Sandewall. A Partial Evaluator, and Its Use as a Programming Tool. *Artificial Intelligence*, 7(4):319-357, 1976.

[Bulyonkov 84]

M.A.Bulyonkov. Polyvariant Mixed Computation for Analyzer Programs. *Acta Informatica*, 21:473-484, 1984.

[Burstall 77]

R.M.Burstall and J.Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44-67, 1977.

[Dixon 71]

J.Dixon. The Specializer, a Method of Automatically Writing Computer Programs. Division of Computer Research and Technology, National Institute of Health, Bethenda, Maryland, 1971.

[Ershov 78]

On the Essence of Compilation. In E.J.Neuhold, editor, *Formal Description of Programming Concepts*, pages 391-420, North-Holland, 1978.

[Ershov 81]

A.P.Ershov. The Transformational Machine: Theme and Variations. In J.Grushka and M.Chytil, editors, *Mathematical Foundations of Computer Science*, Štrbské Pleso, Czechoslovakia, pages 16-32, Lecture Notes in Computer Science, Vol.118, Springer-Verlag, 1981.

[Futamura 71]

Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45-50, 1971.

[Hughes 88]

J.Hughes. Backward Analysis of Functional Programs. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 187-208, North-Holland, 1988.

[Jones 85]

N.D.Jones, P.Sestoft and H.Sondergaard. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In J.-P.Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France*, pages 124-140, Lecture Notes in Computer Science, Vol.202, Springer-Verlag, 1985.

[Jones 86]

N.D.Jones and A.Mycroft. Data Flow Analysis of Applicative Programs Using Minimal Function Graphs. In *Thirteens ACM Symposium on Principles of Programming Languages, St.Petersburg, Florida*, pages 296-306, ACM, 1986.

[Jones 88]

Automatic Program Specialization: A Re-Examination from Basic Principles. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225-282, North-Holland, 1988.

[Mogensen 88]

T.Mogensen. Partially Static Structures in a Self-Applicable Partial Evaluator. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325-347, North-Holland, 1988.

[Ostrovski 88]

B.N.Ostrowski. Implementation of Controlled Mixed Computation in System for Automatic Development of Language-Oriented Parsers. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 385-403, North-Holland, 1988.

[Romanenko 88]

S.A.Romanenko. A Compiler Generator Produced by a Self-Applicable Specializer Can Have a Surprisingly Natural and Understandable Structure. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445-463, North-Holland, 1988.

[Sestoft 86]

The Structure of a Self-Applicable Partial Evaluator. In H.Ganzinger and N.D.Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985*, pages 236-256, Lecture Notes in Computer Science, Vol. 217, Springer-Verlag, 1986.

[Schmidt 86]

D.A.Schmidt. *Denotational Semantics*. Allyn and Bacon, Boston, 1986.

[Sestoft 88]

P.Sestoft. Automatic Call Unfolding in a Partial Evaluator. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485-506, North-Holland, 1988.

[Turchin 72]

V.F.Turchin. Equivalent Transformation of Recursive Functions Defined in Refal. In *Teoriya Yazykov i Metody Programmirovaniya. Trudy Simposiuma*, pages 31-42, Alushta-Kiev, 1972 (in Russian).

[Turchin 79]

V.F.Turchin. A Supercompiler System Based on the Language Refal. *SIGPLAN Notices*, 14(2):46-54, February 1979.

[Turchin 82]

V.F.Turchin, R.M.Nirenberg and D.V.Turchin. Experiments with a Supercompiler. In *1982 ACM Symposium on Lisp and Functional Programming, Pittsburgh, Pennsylvania*, pages 47-55, ACM, 1982.

[Turchin 86]

V.F.Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292-325, July 1986.

[Turchin 88]

V.F.Turchin. The Algorithm of Generalization in the Supercompiler. In D.Bjorner, A.P.Ershov and N.D.Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531-549, North-Holland, 1988.

[Wadler 88]

P.Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming*, Lecture Notes in Computer Science, Springer-Verlag, 1988.