Practical Aspects of Specialization of Algol-like Programs

Mikhail A. Bulyonkov and Dmitrij V. Kochetov

Institute of Informatics Systems, Novosibirsk, Russia

Abstract. A "linearized" scheme of polyvariant specialization for imperative languages is described in the paper. The scheme is intended for increasing efficiency of specialization. Main properties of the scheme are linear generation of residual code and single memory shared by different variants of specialization process.

1 Introduction

As it was mentioned in [12] "The memory requirements for the specializer are at the moment much too high. We have done some work trying to bring it down but there is still a lot that could be done. It seems that these improvements are necessary to make use of the partial evaluator realistic".

To demonstrate the problem let us assume that we specialize an interpreter with respect to a program. Suppose that there is a procedure in the interpreter for interpretation of statements of the source language. When entering the procedure we need to check for different *configurations*, i. e. static memory state in a point of the program being specialized. The number of configurations is obviously the number of statements in the source program, hence it is linear with respect to its size. If we are using a naive approach, we will store all static memory in each configuration, but static memory includes at least the source program representation. So the amount of information to be stored is quadratic with respect to the size of the source program. To make a partial evaluator a useful instrument we should seek for a "more linear" memory requirements.

The process of specialization has very much in common with processes of interpretation and compilation. The transformation performed by specialization to a large degree consists of these processes: when a specializer is designed as a meta-interpreter, it contains a usual interpreter as its part, and for certain input data a specializer functions like a compiler. But the internal behavior of a specializer is different from that of a traditional compiler.

To increase the efficiency of specialization we need to explicate similarity of these processes in a specialization scheme. From our point of view such an explication with respect to compilation means linear generation of residual code which is similar to linear generation of object code. Under the condition we will not need to keep generated parts of the residual program in the memory, and hopefully we will be able to process programs of realistic size. The similarity with interpretation could lie in the usage of a single memory rather than a number of memory instances generated for different variants of specialization process.

The scheme described below has been developed in the course of two projects: partial evaluator for model *MixLan* language, and ongoing *M2Mix* project — a specializer for Modula-2.

2 Specialization Scheme

Our intention is to have a single shared memory and store additionally only differences between variants which are necessary for switching from one variant to another. In order to provide this, we should put some restrictions on the non-determinism of polyvariant specialization. For example, if we start to construct a specialized procedure body as soon as we meet a call to be specialized, the strategy of memorization will be different from that in the case when we postpone residual procedure construction and proceed with the next statement. In fact, we do not argue that the particular scheme described in the paper is the most efficient, but that reasoning on differences between variants makes sense only if the order of processing is specified.

We presume the binding time analysis to have been done, and since dynamic memory is out of interest in this section we will use the term "memory state" instead of "static memory state".

2.1 Specialization of Sequence

Given a sequence of statements $S_1, \ldots, S_n$ and initial memory state M, we start with processing the statement S_1 on the memory state M. The statement under consideration is not necessarily a simple statement — it could be e. g. a conditional or a `while` loop. Generally, processing of the first statement can produce a set of different memory states $M_2^1, M_2^2, \ldots$. Next we will process the second statement S_2 on *all* M_2^i, obtaining a set of memory states $M_3^1, M_3^2, \ldots$ as a result. So the main step of the scheme consists in the processing of a statement on a set of memory states as it is shown in Fig. 1.

Given:
 set of memory states $M_i = \{M_i^1, M_i^2, \ldots\}$
 a statement S_i
$M_{i+1} := 0$
for each $m \in M_i$ do
  process S_i on m resulting new memory states $\{m_1', \ldots, m_n'\}$
  $M_{i+1} := M_{i+1} \cup \{m_1', \ldots, m_n'\}$
od.

Fig. 1. Step of sequence specialization

Following this scheme memory overhead is limited mainly by sets of input and output memory states for *one statement*: in most cases we can discard all

information about a set of input memory states as soon as we have completed processing of the corresponding statement.

Linear traverse of program resembles the *monovariant* scheme (which was called *strict* in [7]) with the difference that each statement is processed many times before processing of the next one. It allows to append generated pieces of a code to residual program immediately. This is a basis for "linear" code generation.

Improvement of the algorithm is based on the observation that we can *statically* determine potential difference between memory states produced by processing of the current statement. Consider the following sequence of statements:

```
m := 1;

if d then k := k*2 else k := k+2;

m := m+k; k := 2;
```

(Here and further on we will assume that variables `k`, `l`, `m` and `n` are static, and all others are dynamic. Also since we concentrate on the static actions, we do not show dynamic "decorations" which can appear at any place). Obviously, the first statement in the example always produces a single memory state. The second statement can produce at most two memory states, but we can be sure that they will differ only in (the value of) variable `k`. When the third statement is processed on a given memory state it produces a single memory state. However, since the statement will potentially be processed on two memory states differing in the variable `k` the resulting memory states will potentially differ in variables `k` and `m`. Finally after execution of the fourth statement only variable `m` can vary.

Let $Stat$, Var, and Val denote the sets of statements, variables, and values respectively. Formally, we can define function

$$\delta : Stat \longrightarrow (2^{Var} \longrightarrow 2^{Var})$$

where $x \in \delta[\![S]\!]v$ means that variation of values of variables from $v \subseteq Var$ *before* processing of S implies that x can potentially have different values *after* processing of S. This function allows to annotate statements in a sequence as follows:

$$\begin{aligned} \Delta[\![S_1]\!] &= \emptyset \\ \Delta[\![S_{i+1}]\!] &= \delta[\![S_i]\!]\Delta[\![S_i]\!] \end{aligned}$$

$\Delta[\![S_i]\!]$ is a set of variables which can have different values *before* processing of S_i in the course of specialization of sequence $S_1; \ldots; S_n$. For the above sequence we have

$$\begin{aligned} \Delta[\![\texttt{m := 1}]\!] &= \emptyset \\ \Delta[\![\texttt{if d then k:=k*2 else k:=k+2}]\!] &= \emptyset \\ \Delta[\![\texttt{m:=m+k}]\!] &= \{\texttt{k}\} \\ \Delta[\![\texttt{k:=2}]\!] &= \{\texttt{m}, \texttt{k}\} \end{aligned}$$

Since values of variables from $\Delta[\![S]\!]$ can vary before repetitive execution of S, at least these values need to be stored in input memory states for S. However restoring values of variables from $\Delta[\![S]\!]$ is not sufficient for switching to the next memory state after processing of the previous one: before each but the first iteration of processing of S it is necessary to restore results $\rho[\![S]\!]$ of statement S. For example, `m:=m+k` changes the value of `m`, and hence the value must be restored. On the other hand restoring all the results of S is redundant: it is sufficient to restore only those results of S_i which are at the same time its arguments, $\alpha[\![S]\!]$. For example, in the following sequence

```
if d then k := k*2 else k := k+2;

l := 2;

m := m+k;
```

`l` is not an argument of the second statement and so it need not be restored. But `m` is an argument of the third statement and its value must be restored before each iteration.

Furthermore, a part of the results of a statement S possibly belongs to $\Delta[\![S]\!]$ and will be restored anyway. Therefore, the set we are looking for is

$$\Gamma[\![S]\!] = (\rho[\![S]\!] \cap \alpha[\![S]\!]) \setminus \Delta[\![S]\!]$$

Let us define "store" and "restore" operations more formally. Let $Domain(f)$ denote the domain of function f. The "store" operation coincides with the functional operator $|$ of narrowing of function domain:

$$(m|D)(x) = \begin{cases} m(x) & \text{if } x \in D \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The "restore" operation coincides with the functional operator $\oplus$ of union of function domains:

$$(m_1 \oplus m_2)(x) = \begin{cases} m_1(x) \text{ if } m_1(x) \text{ is defined} \\ m_2(x) \text{ otherwise.} \end{cases}$$

From this point we will use a *single* global memory M shared by variants of execution and a set of input memory states will represent differences of variants with respect to M. Now we can define more precisely the main step of the scheme (see Fig 2).

Given:
 memory $M : Var \to Val$
 set of memory states $M_i = \{M_i^1, M_i^2, ...\}$
  where $Domain(M_i^j) = \Delta[\![S_i]\!]$
$M_{i+1} := \emptyset$
$\gamma := M|\Gamma[\![S_i]\!]$
for each $m \in M_i$ do
  $M := M \oplus m$
  process S_i on M resulting new memory states $\{m'_1, \ldots, m'_n\}$
  $M_{i+1} := M_{i+1} \cup \{m'_1|\Delta[\![S_{i+1}]\!], \ldots\}$
  $M := M \oplus \gamma$
od.

Fig. 2. Step of sequence specialization using a single memory

2.2 Specialization of Block Statement

Let us extend the language by block statements. By definition, the Δ-annotation of the first statement in any sequence is *always* empty. It means that each iteration in specialization of a block statement on a set of input memory states consists of two actions. First we restore the current memory state, and then we recursively apply the procedure for specialization of a sequence of statements to the body of the block on the *single* memory state. For example, consider the fragment

```
if d then k := k*2 else k := k+2;

begin

  l := l+k; m := l+2

end;

m := 1;
```

with the following annotation

$$\begin{aligned}
&\Delta[\![\texttt{l:=l+k}]\!] &&= \emptyset\\
&\Delta[\![\texttt{m:=l+2}]\!] &&= \emptyset\\
&\Delta[\![\texttt{begin l:=l+k; m:=l+2 end}]\!] &&= \{\texttt{k}\}\\
&\Delta[\![\texttt{m:=1}]\!] &&= \{\texttt{k}, \texttt{l}, \texttt{m}\}
\end{aligned}$$

Had we "unfold" the block, we would be forced to store the various values of `l` and `k` before processing of `m:=l+2`. For the variables which differ after specialization of block $B =$ `begin` $S_1 \ldots S_n$ `end` we have

$$\delta[\![B]\!]v = \delta[\![S_n]\!] \ldots \delta[\![S_1]\!]\ v.$$

If blocks in source language serve also for declaration of local variables, they could be very helpful for a better solution of configuration analysis, since obviously local variables do not appear in configurations outside their scope of declaration. In the following fragment

```
begin
  var k,l;
  begin
    var m;
    if d then m := 1 else m := 2;
    k := m;
  end;
  l := k
end;
```

we have that

$$\Delta[\![\texttt{k:=m}]\!] = \{\texttt{m}\}$$
$$\Delta[\![\texttt{l:=k}]\!] = \{\texttt{k}\}$$

If the inner block were unfolded resulting the statement

```
begin
  var k,l,m;
  if d then m := 1 else m := 2;
  k := m;
  l := k
end;
```

we would have that

$$\Delta[\![\texttt{l:=k}]\!] = \{\texttt{k}, \texttt{m}\}$$

which implies that we need to store values of `m` before processing of `l:=k`, even if `m` is dead at this point. Our experience has shown that in many cases the problem of *dead variables* [11] could be avoided if it were possible to localize dead variables.

2.3 Specialization Points and Configurations

Essentially, the main idea behind block specialization is that specialization of a sequence splitted into blocks may be performed more efficiently then in the case of an unstructured sequence. This is the case when specialization of all statements in a subsequence (except for the last statement) does not lead to several memory states, i. e. for all S

$$\Delta[\![S]\!]\emptyset = \emptyset$$

Such splitting can be performed on pre-processing phase. In the following example

```
m := 1;

if d then k := k*2 else k := k+2;

m := m+k; k := 2;
```

the appropriate splitting would look like

```
begin

  m := 1;

  if d then k := k*2 else k:=k+2

end;

begin

  m:=m+k;

  k:=2

end;
```

This transformation can be considered as an imperative analogue of call unfolding analysis in the context of functional languages. The splitting points in a

sequence are called *specialization points*, also by analogy with terminology used in specialization of functional languages [10].

Now we can return to specialization of sequences. Let us identify a specialization point with a statement following the *generating statement* S such that $\delta[\![S]\!]\emptyset \neq \emptyset$: only processing of generating statements originates several variants. All other statements being processed on a memory variant produce at most one output memory variant. Therefore it is sufficient to keep track of variants at specialization points only. At any specialization point `SP` corresponding memory instance can be reconstructed from the values of variables from $\Delta[\![\texttt{SP}]\!]$. We call a pair consisting of specialization point `SP` and a set of values of variables from $\Delta[\![\texttt{SP}]\!]$ a *configuration*. $\Delta[\![\texttt{SP}]\!]$ is called *the set of configurational variables* of the specialization point `SP`.

2.4 Conditional Statements

Specialization of conditional statement with static test is trivial — it reduces to the specialization of the corresponding branch. For conditional such as $S =$ `if` E `then` S_1 `else` S_2, where E is static, we have[1],

$$\delta[\![S]\!]v = \begin{cases} v \cup \rho[\![S_1]\!] \cup \rho[\![S_2]\!] \text{ if } \alpha[\![E]\!] \cap v \neq \emptyset \\ \delta[\![S_1]\!]v \cup \delta[\![S_2]\!]v. \end{cases}$$

We assume that in the following discussion all conditional statements have dynamic tests. Consider a fragment:

```
k := 0;

l := 0;

if d then k := k+1

else begin k := k*2; l := 3 end;
```

Here we must process *both* branches of the conditional statement. Each branch must be specialized on the same input memory state. Therefore specializer must restore the value of `k` (result of the first branch) before specialization of the second one. Resulting states can differ only in results of the statement branches, i. e. for conditional statement with dynamic test

$$\delta[\![S]\!]v = v \cup \rho[\![S_1]\!] \cup \rho[\![S_2]\!].$$

In the above example `k` and `l` can vary after specialization. The algorithm for specialization of conditional is given on Fig. 3. Note that the algorithm is not symmetrical. If the size of $\rho[\![S_2]\!] \cap \alpha[\![S_1]\!]$ is less than the size of $\rho[\![S_1]\!] \cap \alpha[\![S_2]\!]$ then it would be better to start with specialization of the `else` branch.

[1] For the sake of simplicity we assume that E is free of side effects.

Given:
memory state $M : Var \to Val$
statement $S =$ `if` E `then` S_1 `else` S_2
$\gamma := M|(\rho[\![S_1]\!] \cap \alpha[\![S_2]\!])$
process S_1 on M resulting new memory states $M_1 = \{m_1^2, \ldots\}$
$M := M \oplus \gamma$
process S_2 on M resulting new memory states $M_2 = \{m_1^2, \ldots\}$
specialization of the whole conditional results in $M_1 \cup M_2$

Fig. 3. Specialization of conditional

2.5 Labels and `goto` Statements

Undisciplined usage of `goto`'s actually ruins all our attempts for efficient specialization. So we assume that the source language does not allow such nasty features as jumps into compound statements, or in other words that the scope of a label is limited by the sequence where the label appears.

First of all, we want to distinguish between static and dynamic `goto`'s, i. e. between those which can be performed at specialization time and those which have to be done in residual program. When a `goto` statement is processed, we attempt to interrupt the specialization of all compound statements containing the statement up to the corresponding label level and continue processing from the sequence starting with the label. If interrupted statements require additional actions to complete their specialization, we have to suspend control transfer, i. e. to declare the `goto` statement dynamic. Otherwise the statement is static. In the following example

```
   k := 0;

   if d1 then goto lab else k := 2;

lab:

   d2 := k;
```

`goto lab` is dynamic because conditional statement needs additional actions to complete specialization — processing of the second branch. Therefore, the sequence itself requires analogous steps. Consider another case:

```
   k := 0;

   if d then k := k-1;

   if k>0 then goto lab;
```

```
   .....

lab:
```

Specialization of the second statement generates two configurations. We cannot execute `goto lab` in the first configuration — to complete the sequence specialization a specializer has to process the second configuration. Hence `goto lab` is dynamic,even if it is not explicitly located under dynamic test. On the other hand, in the fragment

```
   k := 0;

   goto lab;

   if d then k := k-1;

   lab:
```

we do not need to suspend the `goto`-statement since at this point only one configuration is possible. Hence we classify the statement as static.

We declare a label `lab` dynamic if there is a dynamic `goto lab` statement. Furthermore, this makes all jumps to the label dynamic. We introduce a specialization point for each dynamic label, since a specializer will potentially restart processing of a sequence starting with this label for different configurations coming from different —tt goto's. Specialization of a `goto` statement adds suspended configuration, as it is shown on Fig. 3.

Given: memory state $M : Var \to Val$ statement $S =$ `goto l` result is configuration $(\texttt{l}, M\|\Delta[\![\texttt{l}:]\!])$

Fig. 4. Specialization of `goto`

After specialization of a sequence we must process all suspended configurations related to the sequence's labels. When no new suspended local configurations appear, we discard all local configurations.

Now we determine the variables which must appear in Δ-annotation of labels and whose values have to be stored in suspended configurations in the case when corresponding label is dynamic. In the following fragment `lab` is static:

```
   if d then k := k+1

   else k := 0;

   if k=0 then goto lab;

   k := 3;

lab:
```

Two configurations will be generated after specialization of the first conditional statement. The fragment starting from the second conditional statement will be processed on both configurations but `k:=3` will be executed during only one configuration processing. Therefore `k` can have different values at the point of label `lab` and must be included in $\Delta[\![\texttt{lab:}]\!]$. In general, for any static label `lab`, $\Delta[\![\texttt{lab:}]\!]$ will contain results of all linear fragments delimited by `lab` itself and a conditional containing jump to `lab` whose test values differ on different execution variants.

Consider the fragment with dynamic label `lab`:

```
   l := 0;

 lab:

   begin

     m := l+k;

     if d then k := k+1 else goto lab;

   end;

   l := k-1;
```

There are two specialization points in this fragment: the one before `begin` and the other after `end`. Specialization of `goto lab` will suspend some configuration which will be processed only after `l:=k-1` statement execution. But to resume processing starting from `lab`, we need the previous value of `l`. Therefore we must store the value of `l` in suspended configuration. The same holds for all results of the subsequence starting from `lab`.

In general in configuration associated with dynamic label located in a sequence `S`, we must store all results of subsequence starting from the first specialization point in `S`, or from the first statement in `S` containing a jump to the

label. The reason is that we can not determine in advance the order (in terms of labels) in which suspended configurations will be generated and processed.

Such efficiency loss can be compensated by the method already discussed, namely more adequate structuring of program and localization of control. The following fragment will be specialized more efficiently despite of semantic equivalence to the previous one:

```
l := 0;

begin

  lab:

    m := l+k;

    if d then k := k+1

    else goto lab

end;

l := k-1;
```

Here `l` is not a result of the block containing `lab`, and there is no need to store it in suspended configurations. The example illustrates that localization of labels can considerably decrease the amount of stored information.

2.6 Loops

In *MixLan* project loops are preprocessing phase objects . They are translated into `goto`'s, conditionals and blocks. For example

```
for i := E1 to E2 do S
```

is translated into

```
begin

  var i : integer;

  i:=E1;

  lab:
```

```
    if i<=E2 then begin S; i:=i+1; goto lab end
end;
```

Therefore specialization of loops is expressed as specialization of `goto` statements. The outer block is very essential: if `lab` is dynamic, then only loop results will be stored in suspended configuration. With such a translation we do not loose both precision of configuration analysis and efficiency of specialization process.

2.7 Procedures

A procedure body can be divided into static and dynamic parts. If the dynamic part of a procedure is empty, we perform a call at specialization time. Otherwise procedure is called *dynamic*. Processing of a dynamic procedure call consists in evaluating output memory state(s) and constructing residual version of the procedure. We postpone the construction step until specialization of the block containing the procedure declaration is completed. Since during block specialization we do not need bodies of residual procedures, no memory is wasted to store them. In order to collect all input memory states of dynamic procedure, we associate with such a procedure a specialization point. When specializer encounters a dynamic procedure call, it generates a *procedural configuration*. Configurational variables of the specialization point include static formal parameters of the procedure and those global procedure arguments, which may be modified in the time interval starting from generation of the configuration and ending at the moment when specializer extracts the configuration for processing. In the worst case, we have to store values of variables which are modified either

1. in the subsequence starting from the first specialization point in the block where the procedure is declared, or
2. in the subsequence starting from the first call of the procedure, or
3. in the body of any other dynamic procedure declared in the same block.

Consider the following example:

```
begin
  var k,l,n;
  procedure p(m)
    begin
      l := m+n; d := k
```

```
    end;

  k := 0; n := 0;

  p(k);

  k := k+1;

  p(l);

end;
```

Specializer will generate two suspended procedural configurations: both configurations contain values of parameter `m` and global variable `k`. Therefore, two residual calls will be constructed: `p_k0_m0` and `p_k1_m0`. Note that we do not store the value of `n` in the configurations.

Summarizing the above discussion we extend the algorithm of block specialization: block's body specialization accumulates the set of configurations for local procedures; when specialization is complete for each configuration it is necessary to construct residual procedure declaration.

To evaluate the set of output memory states for a procedure call we need to execute static part of the procedure. In the case of recursive procedures it potentially leads to non-termination. We can avoid the problem due to the fact that the set of output memory states depends only on the procedural configuration. For each procedural configuration a set of output memory states is collected and reused when specializer encounters the configuration again. Collecting of the set of output memory states requires a fixed point iteration because we could need the set before execution of the procedure body is completed. For example, in the following fragment

```
begin

  var k;

  procedure p(x)

    begin

      if x then

        begin p(x-1); x := x-k; ... end;

      k := 1
```

```
    end;
  k := 2;
  p(y);
  write(k);
end
```

when specializer reaches the call `p(y)`, it constructs residual call of `p_k2`, and then evaluates the set of possible values of `k` after the call. At the point of recursive call `p(x-1)` specializer discovers the same configuration, and it tries to fetch the corresponding set of output memory states which is not evaluated yet.

Let us approach the problem more formally. Specializer maintains mapping of the following type:

$$ProcMap = Cfg \longrightarrow 2^M.$$

Let function

$$\Sigma : Stat \longrightarrow Cfg \longrightarrow ProcMap \longrightarrow ProcMap$$

describe the effect of changing this mapping by processing of statement. In particular, if S is a procedure call then

$$\Sigma[\![\mathtt{p(\ldots)}]\!]\, C\, \pi = fix\ \lambda\, \sigma\, .\, \pi \cup \Sigma[\![\mathrm{B}]\!]\, (\mathtt{p}, [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n])\, \sigma,$$

where B denotes procedure body, $v_1, \ldots, v_n$ are values of configurational variables $x_1, \ldots, x_n$ evaluated on the current configuration C, and operation $\cup$ is defined for functions componentwise. In our example we will obtain that

$$\Sigma[\![\mathtt{p(y)}]\!]\, (\mathtt{p}, [k \mapsto 2])\bot = [(\mathtt{p}, [k \mapsto 2]) \mapsto \{[k \mapsto 1]\}]$$

The approach significantly complicate processing of recursive procedures and requires some extra memory. Had we made all side effects of recursive procedures dynamic, as proposed in [10], the problem would not arise at all. But in many cases such a simplification is unacceptably conservative.

3 Comparison with other analyses

The central issue of the described specialization scheme is *configuration analysis* that determines specialization points and corresponding configurational variables. Configuration analysis is not the only way to reduce memory requirements and it has certain similarities with other analyses.

Configuration analysis singularity consists in the fact that it reflects non-standard semantics of programs, the order of statements processing according to

a particular specialization scheme[2]. This is the reason why we failed to capture the intuitively clear idea of configurational variables with the help of common mod/ref analysis. Below we show that configuration analysis does not coincide with other analyses which are also aimed on reduction of memory consumption.

3.1 Configuration analysis vs. inductive variables analysis

The first one is the inductive variables analysis. A variable is inductive at some program point if has at most one dependence. The following example shows that non-configurational variable is not necessarily inductive.

```
for i := 1 to 10 do

  begin

    if d then k := i else k := i-1;

    (**)

    write(k)

  end
```

Here inductive variable `k` is the configurational one at the point `(**)`.

3.2 Configuration analysis vs. dead variables analysis

Let us compare configuration analysis with dead variables analysis. Consider the fragment:

```
begin

  var k,l,m;

  if d then

    begin k := 1; l := 2 end

  else k := 2;

  m := 2;
```

[2] As a matter of fact, the same is true for binding time analysis.

```
    (**)

    l := m

  end
```

At the point `(**)` dead variable `l` belongs to a set of configurational variables, but alive `m` does not. So configuration analysis can not be expressed as a form of dead variables analysis. On the other hand, there is no need to store dead variables in configurations, and the example shows that configurational analysis as described above should be complemented by a sort of dead variables analysis.

4 Benchmarks

We had to focus on memory requirements partly because of rather tough computing environment: IBM PC with 33MHz processor, 1MB of RAM. In the *MixLan* project both interpreter and specializer (realized as meta-interpreter) are implemented in Scheme. All examples in the paper are written in *MixLan*. The *M2Mix* project is implemented for Modula-2 in C. Here we used the method of generating extension which is also generated in Modula-2. The benchmarks discussed below are obtained using *M2Mix*.

The advantages of our approach are based on three factors

1. choice of specialization points,
2. storing in configurations only configurational variables instead of all static memory, and
3. possibility to discard configurations when moving to the next statement in a sequence.

To measure the influence of factors 2 and 3 we switched off corresponding actions in generation extension and added calculations of the size of dynamically allocated/disposed memory. These modifications in no way effected residual programs. The result of experiments for two programs is shown in Table 1.

The first program, `Splines`, is cubic splines interpolation rewritten from Fortran program provided by R. Glück. The second program, `Match`, is translation of C program for pattern matching provided by L. O. Andersen. In both cases translation was straightforward.

As one could expect, the smallest figures in both cases correspond to *M2Mix* approach and the largest ones — to the naive strategy, when we store all static data in configurations and keep them until the end of residual program generation. So our approach requires 5.56 times less memory in case of `Match` program, and 23.7 times less in case of `Splines`. However the impact of each factor depends on particular program. For the `Match` program we gained more from reducing sizes of configurations, while for the `Splines` program discarding of configurations is more important.

Table1. Memory requirements

	`Splines`		`Match`	
	Discard configurations	Keep configurations	Discard configurations	Keep configurations
Store configurational variables only	156	252	261	493
Store all static memory	2288	3696	707	1450

As for the quality of residual programs, we are quite satisfied with obtained speed-ups: 3 times and 4.99 times for `Splines` and `Match`, resp.

5 Related Works

Many ideas which we realized in these projects are based on the old works of Ershov's group [8, 7, 6]. The works of V. Itkin [9] provided a solid theoretical basis for the foundation of our algorithms. A variety of strategies of mixed computation was proposed by B. Ostrovski [13] for a dialect of Pascal. The idea of linearization of mixed computation first appeared in [4].

The most challenging and stimulating was a breakthrough made by L. O. Andersen in specialization of realistic C programs [2, 10]. In particular, he proposed the method of memorizing procedures side effects. This work re-attracted the interest of PE community to the imperative programming; it was followed by considerable success in specialization of Fortran [5, 3] and object-oriented programs [12].

These works have shown that non-trivial partial evaluation of real-life imperative languages is possible in principal. Our work is an attempt to make it practical.

6 Conclusion

We presented a specialization scheme, which while being truly polyvariant takes care not only about the quality of residual programs, but also about the efficiency of the specialization process itself. We believe that a practical specializer for a language should not impose much more requirements than a compiler and/or an interpreter for the language.

We gave only informal outline of the method: limited size of the paper forced out such important issues as binding time analysis, partially static structures, aliasing, external procedures and data explication. Non-trivial treatment of all these issues is crucial for realization of the scheme.

References

1. Andersen, L.O.: *Partial evaluation of C and automatic compiler generation.* LNCS **641** (1992) 251–257.
2. Andersen, L.O.: *Self-applicable C program specialization.* In Procs. of the Partial Evaluation and Semantics-Based Program Manipulation'92 (1992) 54–61.
3. Baier, R., Glück R., Zöchling, R.: *Partial evaluation of numerical programs in Fortran.* In Procs. of the Partial Evaluation and Semantics-Based Program Manipulation'94 (1994) 119–132.
4. Barzdin, G.Ja., Bulyonkov, M.A.: *Mixed computation and compilation: Linearization and decomposition of a compiler.* Computing Center, Siberian Branch of the USSR Academy of Sciences **Preprint 791** (1988) (In Russian).
5. Blazy, S., Facon, P.: *Partial evaluation for the understanding of Fortran programs.* In Procs. of the Software Engineering and Knowledge Engineering'93 (1993) 517–525.
6. Bulyonkov M.A., Ershov, A.P.: *How do ad-hoc compiler constructs appear in universal mixed computation processes?* In Procs. of the Workshop Partial Evaluation and Mixed Computation (1988) 65–81.
7. Ershov, A.P., Itkin, V.E.: Correctness of mixed computation in Algol-like programs. LNCS **53** (1977) 59–77.
8. Ershov, A.P.: Mixed computation: Potential applications and problems for study. Theor. Comp. Sc. **18** (1982) 41–67.
9. Itkin, V.E.: An algebra and axiomatization system of mixed computation. In Procs. of the Workshop Partial Evaluation and Mixed Computation (1988) 209–224.
10. Jones, N.D., Gomard, C.K., Sestoft, P.: *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.
11. Jones, N.D.: Automatic program specialization: A re-examination from basic principles. In Procs. of the Workshop Partial Evaluation and Mixed Computation (1988) 225–282.
12. Marquard, M., Steensgaard, B.: Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, April 1992.
13. Ostrovski, B.N.: *Implementation of controlled mixed computation in system for automatic development of language-oriented parsers.* In Procs. of the Workshop Partial Evaluation and Mixed Computation (1988) 385–403.

This article was processed using the LaTeX macro package with the LLNCS document class.