Extracting Polyvariant Binding Time Analysis from Polyvariant Specializer

Mikhail A. Bulyonkov *
Institute of Informatics Systems
Russian Academy of Sciences, Siberian Division
Russia
e-mail: mike@isi.itfs.nsk.su

Abstract

Polyvariant binding time analysis allows a program to be transformed for improving propagation and usage of static information. It could be a useful instrument for better specializability. We show that the process of such a transformation is of the same nature as the whole specialization process. Moreover, we present a practical method for realizing polyvariant binding time analysis based on the double application of a polyvariant specializer. The proposed technique is restricted to first order programs with strict semantics.

1 Introduction

The problem of the binding time analysis (hereinafter referred as BTA) is crucial for the design of partial evaluators. This analysis is to find out what part of computation becomes possible when binding times of program arguments are known. There exist two approaches to the implementation of BTA. The first approach is known as an on-line BTA and presumes that accessible data processing is combined with evaluation of binding times of program fragments. When the second approach is used BTA is performed as a separate phase prior to specialization. This is called an off-line BTA. Each of these two approaches has its advantages.

The main advantage of an on-line BTA is that it could be more precise. There are two reasons for that. First, it could take into account particular values of variables. For example, it may happen that the test clause of some conditional always evaluates to true when some program arguments are fixed, and hence the binding time of the whole conditional is determined by the binding time of the then-clause. The other reason is that one and the same program fragment can have different binding times during specialization.

In such situations a traditional monovariant binding time analysis has to use widening since it can not have access to particular values and only one binding time value should be associated with each program fragment: static or dynamic, preferring dynamic in case of ambiguity. However, most of the existing partial evaluators exploit an off-line BTA approach because it proved it can simplify the main course of specialization. It is claimed also that an off-line BTA is crucial for successful self-application of partial evaluator. This claim was undermined by the appearance of self-applicable partial evaluators with an on-line BTA but still it is evident that compilers produced by self-application of such partial evaluators are huge and slow and do not lead to marked improvement of object code [Bon90].

However, the results of such a comparison can be easily explained. A compiler obtained from a partial evaluator with an on-line BTA can perform constant propagation in the program being compiled. This is the justification for its size: it must contain a significant part of the original interpreter. When an off-line BTA is used any operation potentially emerging in object code must be classified as dynamic. The paradox of partial evaluation with an off-line BTA is that it prevents constant propagation in produced compilers while being itself destined to perform constant propagation in a broad sense. As for improvement of object code, the comparison was done on source programs without large constant parts.

Partially this problem is solved by so-called polyvariant BTA [RG92] which allowed a program fragment to be annotated with several binding times. From another point of view a polyvariant BTA copies program points in order to obtain finer annotation to each copy.

Nevertheless the off-line BTA has one undisputable merit. It allows us not to repeat the same evaluation over binding times domain. Actually, when the static/dynamic division of program arguments is fixed, it becomes possible to determine annotations (at least for some fragments) statically. It would be especially useful if we are going to specialize the program many times with one and the same division. But these reasons correspond exactly to those for specialization as the whole. We will try to use this observation for justification and realization of a polyvariant BTA on the basis of a polyvariant partial evaluator.

2 Grounds for polyvariant BTA

The necessity of an off-line BTA for successful self-application is usually explained by the fact that otherwise it is impossible to determine binding times in a program considered as specializer's accessible data. So in realization of Futamura's second projection

$$comp = spec(spec, int)$$

*This work was partially supported by the grant No. 2-15-2-43 from the Ministry of Science, Education and Technical Policy

ACM-PEPM'93-6/93/Copenhagen, DK
© 1993 ACM 0-89791-594-1/93/0006/0059...$1.50

we do not know what will be accessible in the interpreter *int*. This difficulty disappears if it is presumed that the first argument of a specializer is a program annotated by BTA:

$$spec(spec^{ann}, int^{ann})$$

We suggest another approach to the problem which consists in making the binding time information an explicit argument of the specializer

$$spec : P \times Div \times D \to P$$

where Div denotes a set of *program divisions* in the sense of [Jon88]. Recall that program division is a triple of functions $\mu = (\sigma, \delta, \pi)$, where σ extracts static part of a program state, δ extracts dynamic part, and π reconstitutes the whole state from static and dynamic parts (see [Jon88] for details). Then we can formulate that if

$$spec(p, \mu, D) = p^{\mu}_{(\sigma\ D)}$$

then

$$p^{\mu}_{(\sigma D)}\ (\delta\ D) = p\ D$$

Here $p^{\mu}_{(\sigma D)}$ denotes projection of the program p on the static part of data D with respect to program division μ.

Note that with such an approach a specializer has three instead of two arguments. Hence we have to revise Futamura's projections. Let $\mu_0 = (\sigma_0, \delta_0, \pi_0), \mu_1 = (\sigma_1, \delta_1, \pi_1)$, where

$$\begin{array}{lcllcl} \sigma_0(x,y,z) & = & (x,y) & \sigma_1(x,y) & = & x \\ \delta_0(x,y,z) & = & z & \delta_1(x,y) & = & y \end{array}$$

Futamura's second projection will take the form

$$comp = spec(spec, \mu_0, (int, \mu_1, *)) = spec^{\mu_0}_{(int,\mu_1)}$$

since

$$spec^{\mu_0}_{(int,\mu_1)}(p, *) = spec(int, \mu_1, (p, *)) = int^{\mu_1}_{p}$$

Two other projections can be reformulated analogously. Of course we consider only reasonable divisions. So it will not make sense to specify division for a program unknown or to specify data when the division is not given.

Since BTA here is embedded in a specializer it would be natural to attempt to extract it by self-application when a program p and a division $\mu = (\sigma, \delta, \pi)$ are given. So we have

$$spec^{\mu_0}_{(p,\mu)}\ d = spec(p, \mu, d) = p^{\mu}_{(\delta\ d)}$$

Actually, what is obtained here is not an analogue of an annotated program but a generating extension. What we really want to achieve is to transform a program as data from the source representation to some other one, and then pass it together with static data to the residual specializer. This can be done by exploiting the technique of data specialization [Bul91]. In the above notation data specialization can be defined as a couple of mappings

$$\begin{array}{lll} Dmix & : P \times Div \times D & \to D \\ Pmix & : P \times Div & \to P \end{array}$$

such that if

$$\begin{array}{lcl} Dmix(p, \mu, d) & = & [\sigma\ d]^{\mu}_{p} \\ Pmix(p, \mu) & = & p^{\mu} \end{array}$$

then

$$p^{\mu}([\sigma\ d]^{\mu}_{p}, \delta d) = pd$$

Here $[\sigma\ d]^{\mu}_{p}$ denotes a "specialized version of static part of data d with respect to division μ" which are to be used by intermediate program p^{μ}. The symmetry of this notation to that for projection emphasizes the fact that a source program and its static data have exchanged places — results of specialization are concentrated in data rather than in a residual program.

Instantiation $p = spec$, $\mu = \mu_0$, $d = (prg, \mu_{prg}, data)$ implies

$$spec^{\mu_0}([prg, \mu_{prg}]^{\mu_0}_{spec}, data) = prg^{\mu_{prg}}_{data}$$

Note that here we obtained a processor $spec^{\mu_0}$ which does not depend on the source program and realizes the same function as the specializer *spec* when the source program and the division are fixed. The first argument $[prg, \mu_{prg}]^{\mu_0}_{spec}$ of the $spec^{\mu}$ contains all necessary information about the source program and the division and so it can be considered as an analogue of the annotated program — the result of BTA. Now the BTA processor can be defined as $BTA = Dmix^{\mu_0}_{(spec,\mu_0)}$ because

$$Dmix^{\mu_0}_{(spec,\mu_0)}(prg, \mu_{prg}) = [prg, \mu_{prg}]^{\mu_0}_{spec}$$

Notice that we do not assume anything about the processor $Dmix$: it might have either an on-line or an off-line BTA. Actually, a polyvariant $Dmix$ with an off-line monovariant BTA is sufficient for obtaining a polyvariant BTA.

3 Practical approach

Though the above reasoning can serve as a ground for a polyvariant BTA it gives no straightforward way to implementation. There are several stumble points. First, we suppose the existence of a specializer with an on-line BTA. Second, the technique of data specialization (at least as described in [Bul91]) leads to circular data structures as intermediate representation which can not be processed by the existing specializers.

We propose another approach which is much more realistic and allows the effect of a polyvariant BTA to be obtained mostly by the existing tools. Actually we need a specializer with an on-line BTA because it naturally joins binding time and static data processing. We can reach the same by attaching to a source program some additional fragments which evaluate binding times and still do not change the result of the program. We will illustrate our method on programs in the strict statically scoped functional language Scheme [Dyb87].

3.1 Informal description of the method

Let us take as an example the program from [RG92]

```
(define (main a b)
  (list (test a b) (test b a)))
(define (test x y)
  (+ (* x x) (* y y)))
```

Here parameters of the procedure `test` have different binding times in different calls (henceforth we assume `a` to be static and `b` to be dynamic). A monovariant BTA will classify both of these parameters as dynamic and consequently

will do so for both multiplications. So the residual program for `a = 3` will be the following[1]:

```
(DEFINE (MAIN-0 B_0)
  (LIST (+ (* 3 3) (* B_0 B_0))
        (+ (* B_0 B_0) (* 3 3))))
```

Let us now extend the original program in the following way. For each procedure parameter, say `a`, a new parameter `bt-a` is introduced with the intention to hold binding times of the original parameter `a`. So in a procedure call the expression for additional parameters must simulate the evaluation of binding time of the original actual parameter expression. For the above program such an extension will look like:

```
(define (main bt-a a bt-b b)
  (list (test bt-a a bt-b b)
        (test bt-b b bt-a a)))
(define (test bt-x x bt-y y)
  (+ (* x x) (* y y)))
```

Note that this program is equivalent to the original one. Note also that one can regard this program as a specification of the correctness of a monovariant BTA: a variable, say `x`, can be classified as static if only `bt-x` invariantly equals to `'static`.

Now we specialize this program with `bt-a = 'static`, `bt-b = 'dynamic` and get the following residual

```
(DEFINE (MAIN-0 A_0 B_1)
   (LIST (+ (* A_0 A_0) (* B_1 B_1))
         (+ (* B_1 B_1) (* A_0 A_0))))
```

Obviously this program is also equivalent to the original one, but its specialization leads to

```
(DEFINE (MAIN-0-0 B_1_0)
  (LIST (+ 9 (* B_1_0 B_1_0))
        (+ (* B_1_0 B_1_0) 9))))
```

More interesting is the case when the source program is non-linear, *i.e.* when a procedure call appears in an actual parameter expression. Consider for example the program

```
(define (main a b)
  (list (g a b) (g b a)))
(define (g x y)
  (if (zero? x)
      y
      (g (- x 1) (g (- x 1) x))))
```

Here both parameters of the procedure `g` will be declared by a monovariant BTA as dynamic, because of the same reasons as in the previous example. Straightforward specialization will simply instantiate `a` in the body of `main`. In order to apply the same idea we have to construct an expression which evaluates binding times of the expression `(g (- x 1) x)`, if the additional parameter `bt-x` of the procedure `g` refers to the binding time of `x`. We will do it by constructing an auxiliary procedure `bt-g` which evaluates binding time of `g`'s result when given binding times of `g`'s arguments. The semantics of `bt-g` can be specified as

$$G = lfp\ (\lambda\ \varphi\ .\ \lambda xy\ .\ (x \sqcup y \sqcup \varphi(x, \varphi(x, x)))$$

where $\sqcup$ stands for the least upper bound operation on binding times: $S \sqsubseteq D$. The least fixed point (lfp) exists because all of the operation in the right hand side are monotone.

Since the arguments of G range through the finite domain, we can define G as the table

G =

	S	D
S	S	D
D	D	D

Now using this table we can properly define the procedure `bt-g`

```
(define (bt-g x y)
  (if (eq? x 'static)
      (if (eq? y 'static)  'static  'dynamic)
      (if (eq? y 'dynamic) 'dynamic 'dynamic)))
```

and use this definition in the extension of the source program

```
(define (main bt-a a bt-b b)
  (list (g bt-a a bt-b b) (g bt-b b bt-a a)))
(define (g bt-x x bt-y y)
  (if (zero? x)
      y
      (g bt-x
         (- x 1)
         (bt-g bt-x bt-x)
         (g bt-x (- x 1) bt-x x))))
```

Notice that since the procedure `bt-g` is defined in a tabular manner, the specializer will not have to repeat the static computation for finding the least fixed point each time it meets a call to `bt-g`. The result of specialization of the extended program is the following:

```
(DEFINE (MAIN-0 A_0 B_1)
  (LIST
    (IF (ZERO? A_0)
        B_1
        (G-0-2 (- A_0 1) (G-0-2 (- A_0 1) A_0)))
    (IF (ZERO? B_1)
        A_0
        (G-0-4 (- B_1 1) (G-0-4 (- B_1 1) B_1)))))
(DEFINE (G-0-4 X_0 Y_1)
  (IF (ZERO? X_0)
      Y_1
      (G-0-4 (- X_0 1) (G-0-4 (- X_0 1) X_0))))
(DEFINE (G-0-2 X_0 Y_1)
  (IF (ZERO? X_0)
      Y_1
      (G-0-2 (- X_0 1) (G-0-2 (- X_0 1) X_0))))
```

It is easy to see that the monovariant BTA will classify both parameters of the procedure `G-0-2` as static and both parameters of `G-0-4` as dynamic. This solution is exact in the sense that parameters are classified that way in each call and no widening occurs here.

Specialization of the obtained program for `A-0 = 2` yields

```
(DEFINE (MAIN-0-0 B_1_0)
  (LIST 1
        (IF (ZERO? B_1_0)
            2
            (G-0-4-0-2
               (- B_1_0 1)
               (G-0-4-0-2 (- B_1_0 1) B_1_0)))))
```

[1] Residual programs are typed in capital letters as the unchanged output of *Similix* autoprojector [Bon90]

```
(DEFINE (G-0-4-0-2 X_0_0 Y_1_1)
  (IF (ZERO? X_0_0)
      Y_1_1
      (G-0-4-0-2
         (- X_0_0 1)
         (G-0-4-0-2 (- X_0_0 1) X_0_0))))
```

In this final residual program the first component of the result of `MAIN-0-0` has been reduced completely. The only residual version `G-0-4-0-2` of the original procedure g is always called with dynamic arguments.

3.2 Formal description of the method

We restrict our consideration by the simple subset of Scheme. Programs in this subset are sets of procedure definitions. The syntactic classes are

$\texttt{Prg} \in \mathit{Program}$
$\texttt{Def} \in \mathit{ProcedureDefinition}$
$\texttt{P} \in \mathit{ProcedureName}$
$\texttt{E} \in \mathit{Expression}$
$\texttt{O} \in \mathit{BasicOperator}$
$\texttt{C} \in \mathit{Constants}$
$\texttt{V} \in \mathit{Variable}$

The syntax of the language is the following

```
Prg ::= Def+
Def ::= (define (P V...) E)
E   ::= C | V | (if E E E) | (O E...) | (P E...)
```

Binding times form two points abstract domain $BTval = \{S, D\}$[2], where $S \sqsubseteq D$. $X \sqcup Y$ denotes the least upper bound of X and Y. We need two kinds of abstract environments. The first one associates a binding time value with a variable:

$$\rho \in VarAbs = Variable \rightarrow BTval$$

An environment of the second kind associates a binding time function with each procedure name

$$\varphi \in ProcAbs = ProcedureName \rightarrow BTval^* \rightarrow BTval$$

We assume that the function $\mathcal{O}$ associating a monotone binding time function with each basic operator is predefined

$$\mathcal{O} : BasicOperator \rightarrow BTval^* \rightarrow BTval$$

There are two abstract semantic functions: $\mathcal{D}$ for evaluating functions and $\mathcal{E}$ for evaluating expressions in the domain $BTval$

$\mathcal{D} : ProcedureDefinition^+ \rightarrow ProcAbs$
$\mathcal{D}[\![\texttt{(define (P V...) E) ...}]\!] =$
$\quad = lfp(\lambda\varphi.[[\![\texttt{P}]\!] \mapsto \lambda v.\mathcal{E}[\![\texttt{E}]\!]\varphi[[\![\texttt{V}]\!] \mapsto v, \ldots], \ldots]$
$\mathcal{E} : Expression \rightarrow ProcAbs \rightarrow VarAbs \rightarrow BTval$

$$\begin{array}{lcl}
\mathcal{E}[\![\texttt{C}]\!]\varphi\rho & = & S \\
\mathcal{E}[\![\texttt{V}]\!]\varphi\rho & = & \rho[\![\texttt{V}]\!] \\
\mathcal{E}[\![\texttt{(if E}_1\ \texttt{E}_2\ \texttt{E}_3\texttt{)}]\!]\varphi\rho & = & \mathcal{E}[\![\texttt{E}_1]\!]\varphi\rho \sqcup \mathcal{E}[\![\texttt{E}_2]\!]\varphi\rho \sqcup \mathcal{E}[\![\texttt{E}_3]\!]\varphi\rho \\
\mathcal{E}[\![\texttt{(O E}_1 \ldots \texttt{E}_n\texttt{)}]\!]\varphi\rho & = & \mathcal{O}[\![\texttt{O}]\!][\mathcal{E}[\![\texttt{E}_1]\!]\varphi\rho, \ldots, \mathcal{E}[\![\texttt{E}_n]\!]\varphi\rho] \\
\mathcal{E}[\![\texttt{(P E}_1 \ldots \texttt{E}_n\texttt{)}]\!]\varphi\rho & = & \varphi[\![\texttt{P}]\!][\mathcal{E}[\![\texttt{E}_1]\!]\varphi\rho, \ldots, \mathcal{E}[\![\texttt{E}_n]\!]\varphi\rho]
\end{array}$$

The abstract semantics defined this way directly corresponds to the concrete semantic of the language.

[2] Binding times domain could be more complicated, but two points domain is sufficient for our purposes. we consider nonterminating computations and dead code as static Though there would be no problem to add the binding time value X to represent side-effecting evaluation like in [Bon90]

Now for each procedure P of the original program an associated procedure bt-P[3] with the semantic $\mathcal{D}[\![\texttt{P}]\!]$ should be constructed. Let $BTDEF[\![\texttt{P}]\!]$ denote the definition of the procedure bt-P. A straightforward definition based on the above semantic equations will suffice, if only the specializer is able to process it statically. We can guarantee that by using finiteness of the abstract domain which allows bt-procedures be defined in a tabular manner as it was exemplified in the previous section. This would significantly simplify the process of the first specialization — expansion of the source program. The same could be done to obtain procedures realizing abstractions of basic operators. We assume that the procedure bt-U realizes the least upper bound operation $\sqcup$.

The extension of a source program can be defined by two functions: the function $\mathcal{B}$ constructs an expression for evaluating binding times

$\mathcal{B} : Expression \rightarrow Expression$

$$\begin{array}{lcl}
\mathcal{B}[\![\texttt{C}]\!] & = & \texttt{'static} \\
\mathcal{B}[\![\texttt{V}]\!] & = & \texttt{bt-V} \\
\mathcal{B}[\![\texttt{(if E}_1\ \texttt{E}_2\ \texttt{E}_3\texttt{)}]\!] & = & \texttt{(bt-U } \mathcal{B}[\![\texttt{E}_1]\!] \\
 & & \qquad \texttt{(bt-U } \mathcal{B}[\![\texttt{E}_2]\!]\ \mathcal{B}[\![\texttt{E}_3]\!]\texttt{))} \\
\mathcal{B}[\![\texttt{(O E}_1 \ldots \texttt{E}_n\texttt{)}]\!] & = & \texttt{(bt-O } \mathcal{B}[\![\texttt{E}_1]\!] \ldots \mathcal{B}[\![\texttt{E}_n]\!]\texttt{)} \\
\mathcal{B}[\![\texttt{(P E}_1 \ldots \texttt{E}_n\texttt{)}]\!] & = & \texttt{(bt-P } \mathcal{B}[\![\texttt{E}_1]\!] \ldots \mathcal{B}[\![\texttt{E}_n]\!]\texttt{)}
\end{array}$$

and the function $\mathcal{T}$ — joint evaluation

$\mathcal{T} : Expression \rightarrow Expression$

$$\begin{array}{lcl}
\mathcal{T}[\![\texttt{C}]\!] & = & \texttt{C} \\
\mathcal{T}[\![\texttt{V}]\!] & = & \texttt{V} \\
\mathcal{T}[\![\texttt{(if E}_1\ \texttt{E}_2\ \texttt{E}_3\texttt{)}]\!] & = & \texttt{(if } \mathcal{T}[\![\texttt{E}_1]\!]\ \mathcal{T}[\![\texttt{E}_2]\!]\ \mathcal{T}[\![\texttt{E}_3]\!]\texttt{)} \\
\mathcal{T}[\![\texttt{(O E}_1 \ldots \texttt{E}_n\texttt{)}]\!] & = & \texttt{(O } \mathcal{T}[\![\texttt{E}_1]\!] \ldots \mathcal{T}[\![\texttt{E}_n]\!]\texttt{)} \\
\mathcal{T}[\![\texttt{(P E}_1 \ldots \texttt{E}_n\texttt{)}]\!] & = & \texttt{(P } \mathcal{B}[\![\texttt{E}_1]\!]\ \mathcal{T}[\![\texttt{E}_1]\!] \\
 & & \qquad \ldots \mathcal{B}[\![\texttt{E}_n]\!]\ \mathcal{T}[\![\texttt{E}_n]\!]\texttt{)}
\end{array}$$

Finally the extended program can be obtained by replacing each procedure definition `(define (P V...) E)` of the source program by the couple of definitions:

`(define (P bt-V V ...)` $\mathcal{T}[\![\texttt{E}]\!]\texttt{)}$
$BTDEF[\![\texttt{P}]\!]$

4 Correctness and termination

Proving correctness of our methods for the polyvariant BTA relies mostly on the correctness of a specializer being used. Note that an extended program realizes the same computation as an original one because all attached evaluations are terminating and do not infer the result. So even if attached evaluations are not properly related with BTA used by a specializer the worst that might happen is that binding time properties of an intermediate residual program will not be improved. But still it will be equivalent to the original program. Since the basic operations over finite domain $BTval$ are monotone the least fixed point process used for definition of binding time functions terminates. The first specialization (*i. e.* of an extended program) terminates if specialization of an original program terminates also when all arguments are dynamic: again due to the finiteness of the $BTval$ only finite number of copies for each procedure will be produced by a polyvariant specializer.

Unfortunately, this is not true for the second specialization, because unfoldings performed during the first specialization might change termination properties. Consider the following program

[3] We use the prefix bt- for constructing new names assuming that it will not cause name clashes Actually, one can consider bt- as to be an operation on names

```
(define (main a b)
 (list (g a b) (g b a)))
(define (g x y)
  (if (> x y)
      (+ (g (sub1 x) (add1 x))
         (g (add1 y) (sub1 y)))
      0))
```

Similix terminates on this program and falls into infinite specialization on the extended program, when infinite static loop is not guarded by dynamic conditional. But one must admit that termination in the former case is rather occasional.

5 Compatibility with specializer

Actually this method consists in forcing a specializer to expand a source program, although this expansion in general will pessimize the program: several copies of equivalent procedure definitions can appear, while only evaluations on binding times performed. However, a specializer could be clever enough to notice this "cheating", and remove all unnecessary computation from the expanded program. In fact, *Similix* does so to some extent when it transforms dynamic conditional into lambda-abstractions. We can avoid this by inserting in the beginning of each procedure a dummy operations (e.g. `generalize` in *Similix*) on all added parameters and force a specializer to regard them as dynamic.

On the other hand *Similix* has motives of its own for unfolding (especially performed during postprocessing) that potentially change binding times. So for the program in our first example we can reach the desired effect by double specialization of the source program, when the first run is for all arguments being dynamic.

6 Partially static conditionals

Now we try to approach another case which causes widening: that is when alternatives of a conditional with static test have different binding times. For example, the whole conditional

```
(if (> x 0) y 5)
```

is classified by a monovariant off-line BTA (static `x` and dynamic `y`) as dynamic even if `x` is always positive. The problem is that this fact can be discovered only on the basis of particular data which are not accessible to an off-line BTA.

In [RG92] it was stated that in this case it is impossible to improve binding times by any copying of program fragments. In the higher-order environment the *eta-conversion* can be used for the purpose of binding time improvement [Bon90], though no systematic procedure of applying the transformation is known.

The idea that we are going to exploit is to extend a source program by some actions which are performed by a hypothetical on-line specializer and then to pass the extended program to a real (monovariant) specializer. We will presume that the hypothetical on-line specializer works according to the following strategy: when processing an expression it first checks if there is enough static information to calculate the result of the expression. If the information is sufficient it switches itself to completely static mode and emulates the ordinary valuation function. Otherwise the specializer proceeds with subexpressions in "check-first" mode.

We can express this strategy by transforming a source expression `E` into conditional

```
(if (eq?  Q[[E]] 'static) S[[E]] E)
```

where $\mathcal{Q}[\![\texttt{E}]\!]$ is an expression constructed on the basis of `E` and which evaluates binding time of `E` on particular data. The generation of such an expression will be much easier if we presume that a source program had been already annotated by a monovariant BTA. Nevertheless, the value of $\mathcal{Q}[\![\texttt{E}]\!]$ might be different from the annotation assigned to `E` by BTA. Obviously $\mathcal{Q}[\![\texttt{E}]\!]$ itself is static. The expression $\mathcal{S}[\![\texttt{E}]\!]$ is essentially the same as `E` except that all dynamic variables are replaced by an arbitrary constant. It is safe because the evaluation of $\mathcal{S}[\![\texttt{E}]\!]$ is guarded by the condition which guarantees that this constant in fact will never be used. Naturally there is no need to perform the transformation in its full generality: *e. g.* if some expression is already annotated as static we can leave it unchanged. Moreover, it is possible to apply the transformation only at some specified points, e.g. for procedure calls. So we can restrict our consideration by dynamic expressions only and define transformation $\mathcal{Q}$ as

$\mathcal{Q}[\![\texttt{C}]\!]$ = `'static`

$\mathcal{Q}[\![\texttt{V}]\!]$ = `'dynamic`

$\mathcal{Q}[\![\texttt{(if E}_1\ \texttt{E}_2\ \texttt{E}_3\texttt{)}]\!]$ =
```
(if (eq?  Q[[E1]] 'static)
    (if S[[E1]] Q[[E2]] Q[[E3]])
    'dynamic)
```

$\mathcal{Q}[\![\texttt{(O E}_1 \ldots \texttt{E}_n\texttt{)}]\!]$ = `(bt-O` $\mathcal{Q}[\![\texttt{E}_1]\!] \ldots \mathcal{Q}[\![\texttt{E}_n]\!]$`)`

where `bt-O` is a binding time abstraction of a primitive operator `O`.

To define the transformation for a procedure call we should take care about procedure definitions. For a procedure `P` which is not completely static (for the sake of notational simplicity, suppose that it has two arguments: static `a` and dynamic `b`)

```
(define (P a b) E)
```

we construct two additional procedures of one parameter `a`: the procedure `dbt-P` for evaluation of binding time and the procedure `static-P` for static evaluation:

```
(define (dbt-P a) Q[[E]])
(define (static-P a) S[[E]])
```

Note that the procedure `dbt-P` differs from `bt-P` described earlier. Although both of them result in binding time domain the former is supplied with concrete data while the latter works on abstract values.

Now we can define that

$\mathcal{Q}[\![\texttt{(P E}_1\ \texttt{E}_2\texttt{)}]\!]$ = `(dbt-P` $\mathcal{Q}[\![\texttt{E}_1]\!]$`)`

$\mathcal{S}[\![\texttt{(P E}_1\ \texttt{E}_2\texttt{)}]\!]$ = `(static-P` $\mathcal{S}[\![\texttt{E}_1]\!]$`)`

Let us demonstrate how this transformation works on a typical example. Consider a fragment of an interpreter for evaluating expressions:

```
(define (expr e env)
   (cond
     ((isCst? e) (fetch-Cst e))
     ((isVar? e) (lookup (fetch-Var e) env))
     ((isOp? e) (apply-Op
                  (fetch-Op e)
                  (expr* (fetch-Args e) env)))))
 (define (expr* e* env)
    (if (null? e*)
        '()
        (cons (expr (car e*) env)
              (expr* (cdr e*) env))))
```

Naturally `e` and `e*` are static, `env` is dynamic. Hence the "dynamicity" of `env` propagates backward through `lookup`, `cond`, `expr`, `if`, `expr*`, `cons` and `apply-Op`.

Specialization of expr with respect to the source expression $(x + 3) * (7 - 2)$ will produce a fragment of residual program

```
(apply-Op '*
  (cons
    (apply-Op '+
      (cons
        (lookup 'x env)
        (cons 3 '())))
    (cons
      (apply-Op '-
        (cons 7 (cons 2 '())))
      '())))
```

Here

```
    (cons
      (apply-Op '-
        (cons 7 (cons 2 '())))
      '())
```

was not reduced because apply-Op and cons were classified as dynamic.

Let us now apply the above transformation to each procedure call. So we have

```
(define (expr e env)
  (cond
    ((isCst? e) (fetch-Cst e))
    ((isVar? e) (lookup (fetch-Var e) env))
    ((isOp? e)
      (apply-Op
         (fetch-Op e)
         (if (eq? (dbt-expr* (fetch-Args e))
                  'static)
             (static-expr* (fetch-Args e))
             (expr* (fetch-Args e) env))))))
(define (dbt-expr e)
  (cond
    ((isCst? e) 'static)
    ((isVar? e) 'dynamic)
    ((isOp? e)
      (bt-apply-Op
        'static
        (dbt-expr* (fetch-Args e))))))
(define (static-expr e)
  (cond
    ((isCst? e) (fetch-Cst e))
    ((isVar? e) (lookup (fetch-Var e) '***))
    ((isOp? e)
      (apply-Op
        (fetch-Op e)
        (static-expr* (fetch-Args e))))))
(define (expr* e* env)
   (if (null? e*)
       '()
       (cons
         (if (eq? (dbt-expr (car e*)) 'static)
             (static-expr (car e*))
             (expr (car e*) env)
         (if (eq? (dbt-expr* (cdr e*)) 'static)
             (static-expr* (cdr e*))
             (expr* (cdr e*) env))))
(define (dbt-expr* e*)
   (if (null? e*)
       'static
       (bt-cons (dbt-expr (car e*))
                (dbt-expr* (cdr e*)))))
(define (static-expr* e*)
   (if (null? e*)
       '()
       (cons (static-expr (car e*))
             (static-expr* (cdr e*)))))
```

In the extended version we have two variants of apply-Op: a dynamic variant in the procedure eval and a static one in the procedure static-eval. So specialization of the extended version of eval with respect to the same expression will yield a better object code

```
(apply-Op '*
  (cons
    (apply-Op '+
      (cons
        (lookup 'x env)
        '(3)))
    '(5)))
```

7 Related works and conclusion

Polyvariant BTA for partial evaluation appears in [Cons89]. Definition of abstract semantics for binding times domain, very close to that given in the paper can be found, e.g., in [Laun91]. Some ad hoc technique for a polyvariant BTA was proposed in [RG92] as an extension of a monovariant BTA in the context of the *Similix* partial evaluator. Our approach is not related to any particular partial evaluator and exploits specialization for realization and justification of the program expansion. In contrast to [RG92] we do not have to restart BTA after each unfolding. But still the applicability of our method need to be extended to higher order programs and partially known data structures.

We consider as an advantage of our approach that it is (at least relatively) independent of the specializer. Moreover it allows variations of characteristics of compilers produced from interpreters: if we need a fast and small compiler an original interpreter is used, otherwise, if the main objective is the residual code quality then we exploit various extensions of the interpreter.

8 Acknowledgments

I am grateful to Olivier Danvy and Anders Bondorf — the authors of the *Similix* autoprojector — for their help in coping with the system and porting it to IBM PC that made it possible to verify the idea by exciting experiments. It is my pleasure to thank referees for corrections and suggestions for improvement.

References

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation.* Ph.D. Thesis, University of Copenhagen, Denmark, 1991.

[Bul91] M. A. Bulyonkov. From Partial Evaluation to Mixed Computation. In *Images of Programming* (1991), D. Bjørner, V. Kotov, Eds., North-Holland, pp. 47-60.

[Cons89] Ch. Consel. *Analyse de programmes, Evaluation partielle et Génération de compilateurs.* Ph.D. thesis, LITP, University of Paris 6, France, 1989.

[Dyb87] R. K. Dybvig. *The SCHEME Programming Language.* Prentice-Hall, New-Jersey, 1987.

[Jon88] Automatic program specialization: a re-examination from basic principles. In *Partial Evaluation and Mixed Computation* (1988), D. Bjørner, A. Ershov and N. Jones, Eds., North-Holland, pp. 225-282.

[Laun91] J. Launchbury. *Projection factorisations in partial evaluation.* Cambridge University Press, 1991.

[RG92] B. Rytz, M. Gengler. A Polyvariant Binding Time Analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, 1992.