Binding-Time Analysis in Partial Evaluation: One Size Does *Not* Fit All

Niels H. Christensen, Robert Glück, and Søren Laursen

DIKU, Department of Computer Science
University of Copenhagen, Universitetsparken 1
DK-2100 Copenhagen, Denmark
Email: {mrnc,glueck,scrooge}@diku.dk

Abstract. Existing partial evaluators usually fix the strategy for binding-time analysis. But a single strategy cannot fulfill all goals without leading to compromises regarding precision, termination, and code explosion in partial evaluators. Our goal is to improve the usability of partial evaluator systems by developing an adaptive approach that can accommodate a variety of different strategies ranging from maximally polyvariant to entirely uniform analysis, and thereby make offline specialization more practical in a realistic setting. The core of the analysis has been implemented in FSpec, an offline partial evaluator for a subset of Fortran 77.

1 Introduction

Partial evaluation of imperative programs was pioneered by Ershov and his group [13, 7]; later Jones et al. [21] introduced binding-time analysis (BTA) to achieve self-application of a partial evaluator. This offline approach to partial evaluation has been studied intensively since then.

However, not much attention has been paid to the properties of the binding-time analysis in offline partial evaluation (notable exceptions are [11, 23, 8, 6]). This is surprising because the annotations a BTA produces, *guide the specialization process* of an offline partial evaluator and, thus, control the quality of the program transformation. The choice of the annotation strategy is therefore the *most decisive factor* in the design of an offline partial evaluator.

Existing offline partial evaluators *fix* a particular binding-time strategy (e.g., [3, 9, 12, 22]). None of them allow the partial evaluator to function with different levels of precision, and all systems implement different strategies based on decisions taken on pragmatic grounds. The growing importance of non-trivial applications with varying specialization goals (e.g. interpreter specialization vs. software maintenance) motivated us to examine a more flexible approach to binding-time analysis for imperative languages. Our goal is to improve the usability of partial evaluation systems by developing an analysis framework that allows an easy adaptation and control of different binding-time strategies within the same specialization system.

D. Bjørner, M. Broy, A.V. Zamulin (Eds.): Perspectives of System Informatics. Proceedings.
LNCS 1755, pp. 80–92, 2000.

Program	Source code	Res. code (unif. BTA)	Res. code (poly. BTA)

Monitor
 	
`Upd = FALSE`
`Val = 100`
`OutVal = 0`
`CurVal` *is dynamic*
	1
	`...`
	`10: IF Upd=TRUE THEN`
	`11:  Val:=CurVal;`
	`12: ENDIF;`
	`13: OutVal:=f(Val);`
	`14: OUTPUT OutVal;`
	`...`
		1A
		`...`
		`10: Val:=100;`
		`11: OutVal:=f(Val);`
		`12: OUTPUT OutVal;`
		`...`
			1B
			`...`
			`10: OUTPUT 5;`
			`...`

Affine
 	
`a = 2`
`b = 5`
`x` *is dynamic*
`count` *is dynamic*
	2
	`...`
	`10: IF a>0 THEN`
	`11:  p(x);`
	`12:  GOTO 10;`
	`13: ENDIF;`
	`...`
	`100: PROCEDURE p(y):`
	`101: a:=a-1;`
	`102: b:=b+y;`
	`103: count:=count+1;`
	`104: RETURN;`
		2A
		`...`
		`10: b:=5;`
		`11: p(x);`
		`12: p(x);`
		`...`
		`100: PROCEDURE p(y):`
		`101: b:=b+y;`
		`102: count:=count+1;`
		`103: RETURN;`
			2B
			`...`
			`10: p1(x);`
			`11: p2(x);`
			`...`
			`100: PROCEDURE p1(y):`
			`101: b:=5+y;`
			`102: count:=count+1;`
			`103: RETURN;`
			`104: PROCEDURE p2(y):`
			`105: b:=b+y;`
			`106: count:=count+1;`
			`107: RETURN;`

Fig. 1. Problem source: One BTA is *not* best for all source programs

In this paper we examine the design space of binding-time strategies and develop a framework to formalize different strategies that allows a partial evaluator to function with different levels of granularity. We claim that it is expressive enough to cover all existing strategies and allows the design and comparison of new strategies. The core of the analysis engine is implemented for FSpec, an offline partial evaluator for a subset of Fortran 77 [22]. We assume familiarity with the basic notions of offline partial evaluation, e.g. [19, Part II].

2 Problem Source: One Size Does *Not* Fit All

In existing partial evaluators, the strategy of the binding-time analysis (BTA), and thus its precision, is fixed at design-time; in essence assuming 'One Size Fits All'. The most popular strategy for BTA, due to its conceptual simplicity, is to annotate programs using uniform divisions [19]. In this case *one* division is valid for *all* program points. A polyvariant BTA allows *each* program point to be annotated with *one or more* divisions.

Figure 1 shows two pieces of source programs and for each the result of two different specializations: One directed by a *uniform BTA* (column A) and one directed by a *polyvariant BTA* (column B). We assume *polyvariant program point specialization* [7, 19] (a program point in the source program may be specialized wrt. different static stores). Program *Monitor* updates variable `Val` depending on the value of flag `Upd` (we assume that function `f` has no side effects and that `f(100) = 5`). Program *Affine* repeatedly calls procedure `p`. Variables `a`, `b` and `count` are global.

For *Monitor*, the polyvariant BTA (1B) clearly achieves the best specialization because result `5` is computed at specialization time. The uniform BTA (1A) must consider `Val` dynamic and can therefore not allow the call of `f` to

be computed at specialization time. For *Affine*, the uniform BTA (2A) seems to provide a better specialization. The polyvariant BTA (2B) recognizes that the value of `b` is sometimes static (in the first round of the loop) and creates an extra instance of procedure `p`. This leads to undesirable duplication of code (which is more dramatic for larger programs). Almost all existing partial evaluators, such as C-Mix [3] and FSpec [22], give (1A,2A); Tempo [12] gives (1A,2B).

To conclude, the uniform BTA is preferable for *Affine* and the polyvariant BTA is preferable for *Monitor*. A partial evaluator that is confined to one of the two strategies, A or B, may not be suitable for the task at hand. In such a case the user has to resort to *rewriting the source program* to influence the specialization. This is why we are looking for a more elegant and flexible solution to BTA.

3 Binding-Time Analysis and Maximal Polyvariance

First, we give a quite abstract definition of a programming language as a state transition system. Then, we give a formalization of binding-time analyses and define maximal polyvariance.

3.1 Preliminary Definitions

We consider only first-order deterministic programming languages, and assume that any program has a set of *program points*. Examples include labels in a flow chart language and function names in a functional language. Their essential characteristics is that computation proceeds sequentially from program point to program point by execution of a series of *commands*, each of which updates a program *state*. These states are usually described by a pair consisting of a *program point* and a *store*. The meaning of each command is then a state transformation computing the effect of the command on a state. We assume a small steps semantics (i.e., the execution of each command terminates).

Definition 1. *A* programming language *is a tuple* $L = (\mathcal{P}, \mathcal{C}, \mathcal{S}, [\![\cdot]\!])$*, where* $[\![\cdot]\!] : \mathcal{C} \to \mathcal{S} \to \mathcal{P} \times \mathcal{S}$ *is a partial function. Terminology:* $\mathcal{P}$ *is the set of* program points*,* $\mathcal{C}$ *is the set of* commands*,* $\mathcal{S}$ *is the set of* stores*, and* $[\![\cdot]\!]$ *is the* semantics *of* L*. A* state *is a pair* $(p, \sigma) \in \mathcal{P} \times \mathcal{S}$*.*

Definition 2. *Let* L *be a programming language, then an* L-program *is a partial mapping* $P : \mathcal{P} \to \mathcal{C}$*, where* $\mathcal{P}$ *is the set of program points of* L *and* $\mathcal{C}$ *is the set of commands of* L*. We assume each* L*-program* P *has the property that* $\forall \sigma \in \mathcal{S}. \forall p \in dom(P) : [\![P(p)]\!]\sigma = (p', \sigma')$ *implies* $p' \in dom(P)$*, if defined. Notation: The* initial program point *of a program* P *is denoted by* p_0*.*

Definition 3. *Let* P *be an* L*-program, define* computation step *as transition relation* $\to \subseteq (\mathcal{P}, \mathcal{S}) \times (\mathcal{P}, \mathcal{S})$ *such that* $(p, \sigma) \to (p', \sigma')$ *iff* $[\![P(p)]\!]\sigma = (p', \sigma')$ *is defined. A* computation *(from* $\sigma_0 \in S$*) is a finite or infinite sequence*

$$(p_0, \sigma_0) \to (p_1, \sigma_1) \to \ldots$$

From now on we look at programming languages where the store is modelled by a finite function $\sigma = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ which maps variables $x \in \mathcal{X}$ to values $v \in \mathcal{V}$. We assume there are only finitely many variables in any given program. Notation $\sigma(x_i)$ denotes value v_i in σ. More complicated store models exist and can be handled in our framework (e.g., including locations for modelling pointers and aliasing), but are omitted for simplicity.

3.2 Abstract Formulation of Binding-Time Analysis

The main feature of offline partial evaluation [19] is that program specialization proceeds in two steps: a *binding-time analysis* (BTA) followed by a *specialization phase*. First, the source program is analyzed over a domain consisting of two abstract values, S and D, where S (static) represents a value known at specialization time, D (dynamic) represents a value that may be unknown at specialization time (such a classification of the variables is often called a *division*). Second, the source program is specialized wrt. known values following the static/dynamic annotations made by the BTA.

The BTA associates with each program point one or more *binding-time stores* where each binding-time store maps variables to binding-time values. We limit ourselves to a finite description of binding-time values.

Definition 4. *A* binding-time value *is a value $b \in \mathcal{B}$ where $\mathcal{B} = \{S, D\}$. A* binding-time store *$\beta : \mathcal{X} \to \mathcal{B}$ maps variables to binding-time values. A* binding-time semantics *$[\![\cdot]\!]_{bta} : \mathcal{C} \to (\mathcal{X} \to \mathcal{B}) \to (\mathcal{X} \to \mathcal{B})$ maps a command and a binding-time store to a binding-time store. A* binding-time state *is a pair (p, β), where p is a program point and β is a binding-time store. Notation $\sigma|_{\beta:S}$ denotes a store restricted to variables mapped to S in β.*

Defining binding-time stores as a map from variables to binding-time values does not exclude data structures, such as arrays or records, where the size of the structure is fixed at compile time. For example, fields of a record can be treated as an individual variables. Often a single variable is used to represent the binding-time value of the whole array.

Definition 5. *Let P be an L-program, define* binding-time step *as transition relation $\overset{bta}{\to} \subseteq (\mathcal{P}, \mathcal{B}) \times (\mathcal{P}, \mathcal{B})$ such that $(p, \beta) \overset{bta}{\to} (p', \beta')$ iff*

$$[\![P(p)]\!]_{bta}\beta = \beta' \ \wedge \ \exists \sigma, \sigma'.(p, \sigma) \to (p', \sigma')$$

We expect $[\![\cdot]\!]_{bta}$ to be a realization of the *congruence rules* of language L [19]. Given $[\![P(p)]\!]_{bta}\beta = \beta'$, we expect that for any transition $(p, \sigma) \to (p', \sigma')$, the values of the variables classified as S in β' must be computable from the values of the variables classified as S in β. This congruence requirement is captured more formally by the following definition.

Definition 6. *A binding-time semantics $[\![\cdot]\!]_{bta}$ is* congruent *iff for every program P, any variable $x \in \mathcal{X}$, any transition $(p, \beta) \overset{bta}{\to} (p', \beta')$, and any two stores σ, σ' such that $[\![P(p)]\!]\sigma = (p', \sigma_1)$ and $[\![P(p)]\!]\sigma' = (p', \sigma_2)$ we have*

$$\sigma|_{\beta:S} = \sigma'|_{\beta:S} \wedge \beta'(x) = S \Rightarrow \sigma_1(x) = \sigma_2(x)$$

3.3 Maximally Polyvariant Binding-Time States

The task of a BTA is, given an L-program P and a bt-state (p_0, β_0) of P, to compute a *set of bt-states* (denoted by Ann). This set is always finite because a program has finitely many variables and there are finitely many bt-values. To keep our discussion language-independent, we shall clearly separate the set of bt-states from the syntactic annotation of a source program.

We wish to specify a soundness condition for an annotation, intuitively stating that a specializer should be able to partially evaluate the source program using the annotation. The definition must thus be relative to the specializer. We let the properties of this specialiser be reflected by a corresponding binding-time semantics (which thus represents the needs of the specializer).

Definition 7. *A set of bt-states is called an* annotation. *An annotation, Ann, is sound iff the initial bt-state $(p_0, \beta_0) \in Ann$ and for all $(p_j, \beta_j) \in Ann$ we have*

- *There is a $(p_i, \beta_i) \in Ann$ and a bt-store β'_j such that $(p_i, \beta_i) \overset{bta}{\rightarrow} (p_j, \beta'_j)$ and $\beta'^{-1}_j(D) \subseteq \beta_j^{-1}(D)$.*
- *For all $p_k \in \{p \in \mathcal{P} \mid \exists \sigma, \sigma' : (p_j, \sigma) \rightarrow (p, \sigma')\}$ there is a $(p_k, \beta_k) \in Ann$ and a binding-time store β'_k such that $(p_j, \beta_j) \overset{bta}{\rightarrow} (p_k, \beta'_k)$ and $\beta'^{-1}_k(D) \subseteq \beta_k^{-1}(D)$.*

Definition 8. *Let P be an L-program and let β_0 be an initial bt-store, then $polymax(P, \beta_0)$ denotes the set of bt-states defined by*

$$polymax(P, \beta_0) \overset{\text{def}}{=} \{(p, \beta) \mid (p_0, \beta_0) \overset{bta}{\rightarrow}{}^{*} (p, \beta)\}$$

Clearly, this set is sound. We call it the maximally polyvariant *annotation.*

We have not discussed how to model procedures and calls. This is possible but requires non-trivial extensions of the store model (e.g. locations) which we shall not describe here.

4 Dimensions of Binding-Time Analysis

Programs can be annotated in many ways. A binding-time strategy for realistic applications has to accommodate three important, but—unfortunately—often conflicting transformation goals:

1. **Increasing staticness** by more precise analysis.
2. **Taming code explosion** by reducing the amount of polyvariance at specialization time.
3. **Ensuring termination** of the specialization process by dynamizing operations that lead to infinite transformations.

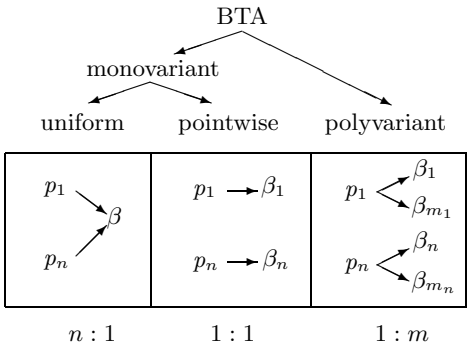

Fig. 2. Granularity of binding-time analysis

A uniform BTA computes *one* division that is valid for *all* program points (illustrated in Fig. 2). For small programs this assumption is reasonable, but not for larger applications because of the non-locality of binding-time effects, a problem with flow-insensitive analyses known from the design of optimizing compilers. For example, a uniform BTA carries the dynamization of a variable in one region to all other regions of a program, even though the variable may serve locally distinct purposes in each region.

Pointwise and *polyvariant* analyses are flow-sensitive. They allow each program point to be annotated with *one or more* local divisions (Fig. 2). This can significantly improve staticness in programs and avoid the need for manual binding-time improvements. For example, the BTA of Tempo [17] computes pointwise divisions for basic blocks and polyvariant divisions on the procedure level.

Increased staticness in a program does not always come for free. *Non-termination* of the specialization process and *code explosion* of the generated programs are some of the risks one faces. In particular, static values that vary without bound, lead to infinite specialization (for each static store encountered at a program point, a specialized version is produced by the specialization phase). Termination can be ensured by dynamizing such static variables. Strategies for ensuring termination without being overly conservative are a topic of current research [5, 14].

5 Strategy Language

Informally, a BTA strategy is a guiding principle for annotation. Known strategies include uniform analysis and pointwise analysis. Our aim is to specify a high-level 'strategy language' which may be used to control a binding-time analysis. The ambition is that the language be simple while offering a large design space allowing to compare the relative strength of different BTA strategies.

Formally, we define a strategy to be a criterion for being *well-formed* (wrt. the strategy). For instance, an annotation is well-formed wrt. the uniform BTA

$$\begin{array}{|ll|}\hline \mathcal{S}_{uniform} & \equiv \beta'(x) = D \\ \mathcal{S}_{pointwise} & \equiv \mathcal{S}_{uniform} \wedge p = p' \\ \mathcal{S}_{polymax} & \equiv \mathit{False} \\ \hline \end{array}$$

Fig. 3. Three well-known BTA strategies

Source code	Annotations
`10: IF Upd=TRUE THEN`	$\langle S, S, D, S\rangle$
`11:  Val:=CurVal;`	$\langle S, S, D, S\rangle$
`12: ENDIF;`	$\langle S, D, D, S\rangle$
`13: OutVal:=f(Val);`	$\langle S, S, D, S\rangle$ $\langle S, D, D, S\rangle$
`14: OUTPUT OutVal;`	$\langle S, S, D, S\rangle$ $\langle S, D, D, S\rangle$

Fig. 4. Polyvariant annotation of *Monitor*: $\langle$`Upd`, `Val`, `CurVal`, `OutVal`$\rangle$

strategy if and only if every variable has the same annotation in all bt-stores in the annotation. In this paper, all strategies are of the form

$$\begin{array}{|c|}\hline \forall x \in \mathcal{X}.\forall (p,\beta),(p',\beta') \in \mathit{Ann}: \\ \mathcal{S}(x,p,p',\beta,\beta') \Rightarrow \beta(x) = D \\ \hline \end{array}$$

where the predicate $\mathcal{S}$ can take many forms. We will identify a strategy with the predicate $\mathcal{S}$ that defines it. We implicitly assume that all annotations be sound. For convenience, we omit the parameters of a predicate, as in the definitions in Fig. 3. Regard the definition of $\mathcal{S}_{uniform}$. This predicate defines a strategy that allows only one annotation for each variable in the source program. The predicate is so simple that it does not need to refer to p or p'.

To see what this strategy means, consider the *Monitor* program. A polyvariant annotation is given in Fig. 4. This annotation is not well-formed wrt. $\mathcal{S}_{uniform}$ since `Val` has more than one annotation. More formally, choosing

$$x = \texttt{Val};\ (p,\beta) = (13, \langle S, S, D, S\rangle);\ (p',\beta') = (13, \langle S, D, D, S\rangle)$$

we evidently get a counterexample to $\mathcal{S}_{uniform}$. We say that x and (p,β) form a *violation* of the strategy. Of course, if an annotation is not well-formed wrt. some strategy $\mathcal{S}$, a violation of $\mathcal{S}$ must exist.

A natural annotation that *does* satisfy the uniformity constraint is the set $\{(p, \langle S, D, D, S\rangle) \mid p \in \{10, 11, 12, 13, 14\}\}$, which is also the one that we would expect as output of a uniform BTA. Note, however, that classifying *all* variables dynamic at *all* program points is an annotation that is also (trivially) well-formed wrt. $\mathcal{S}_{uniform}$. This annotation will be well-formed wrt. any strategy.

Another example of a well-known strategy is $\mathcal{S}_{pointwise}$ which is also defined in Fig. 3. It is obtained by applying the uniform strategy to individual program points, merging bt-stores only if different ones occur at a single point in the program. This strategy forces a monovariant (but not necessarily uniform) annotation of all variables. Finally, as we have implicitly required all annotations to be sound, we get a maximally polyvariant strategy by adding no further requirements.

BTA Classification Formulating the BTA strategies in our language allows us to study their relative strengths formally. Recall the overall classification of BTA strategies given in Figure 2. Most systems (e.g. C-Mix [3] and FSpec [22]) use some BTA strategy, $\mathcal{S}$, which is no stronger than the pure, uniform strategy in the sense that $\mathcal{S} \Rightarrow \mathcal{S}_{uniform}$, i.e. these systems detect no more staticness than one does with the pure, uniform strategy. For several systems biimplication does not hold because some variables are generalized in order to ensure termination of the specializer.

The Tempo system has a non-uniform BTA strategy, $\mathcal{S}_{Tempo}$, that allows a limited amount of polyvariance[1]. Among $\mathcal{S}_{uniform}$ and $\mathcal{S}_{Tempo}$ neither left- or right-implication holds; Tempo is non-uniform, but it forces generalization of static variables defined under dynamic control.

The strategy $\mathcal{S}_{polymax}$ is trivially stronger than all other strategies. Note that what we compare is *precision*, which is *not* a universal measure of quality. We argue that for some source programs the user needs a precise analysis, for other source programs a less precise analysis is better.

6 Simple Construction of Well-formed Annotations

In this section, we take a small detour to sketch one way of implementing a BTA algorithm that is able to realize any strategy as described above. The algorithm builds upon a maximally polyvariant BTA as defined in Section 3. The authors have implemented a maximally polyvariant `PolyMax` function [10] for a non-trivial subset of Fortran—the subset of the FSpec partial evaluator [22].

We assume to have a command $dyn_{p,x}$ in the source language, for which

$$[\![dyn_{p,x}]\!]\sigma = (\sigma, p)$$
$$[\![dyn_{p,x}]\!]_{\mathrm{BTA}}\beta = \beta[x \mapsto D]$$

Also, we assume that for any program P we have $dom(P) \neq \mathcal{P}$ so that we may choose a *new* program point $p_{new} \in \mathcal{P} \setminus dom(P)$.

Our algorithm can be seen in Figure 5. The basic idea is to start out with a maximal annotation, then remove strategy violations one at a time until none are left. Violations are removed by inserting *dyn* commands in the source program where generalization is needed. A maximally polyvariant BTA should be used in step 1 to get maximal preciseness within the constraints defined by the strategy.

Termination of the algorithm is guaranteed by the fact that *Ann* will be strictly increasing its dynamicity during each iteration. Well-formedness (wrt. the input strategy) of the final result should be evident. We do not claim this algorithm to be efficient, it merely serves to illustrate how a strategy chosen at specialization-time can be used to control the result of partial evaluation.

[1] In Tempo, the annotated program may contain several copies of each function – one for each bt-store it is called with. But within one copy, a statement can have only one annotation.

Given source program, P, and a strategy, $\mathcal{S}$:

1. Compute Ann:=`PolyMax`(P).
2. If Ann is well-formed wrt. $\mathcal{S}$ then stop, outputting Ann; else goto 3.
3. Pick $x \in X$ and $(p_{vio}, \beta) \in Ann$ that form a violation of $\mathcal{S}$.
4. Choose a new program point p_{new}, set

$$P := P[p_{vio} \mapsto dyn_{p_{new},x}; p_{new} \mapsto P(p_{vio})]$$

5. Goto 1.

Fig. 5. Algorithm implementing BTA parametrized by strategy

7 An Example Strategy

To illustrate our method, we show a new strategy that can be modeled in our framework. It is characterized by separate treatment of different language constructs, e.g. conditionals, loops and procedures.

The idea is to minimize code explosion in the residual program while being robust wrt. procedure inlining[2], a feature that is not currently achieved by any system implementing polyvariant procedure calls for an imperative language. We also wish to allow polyvariance elsewhere as long as it can only lead to code explosion in the annotated program – not the residual program. Towards this end, we decree that loop entry points may only be annotated polyvariantly if the test-expression (i.e. the loop condition) is static, in which case only one branch will be chosen by the specializer (leaving the other branch as dead code in the annotated program). We denote by $\mathcal{P}_{loopentry}$ the set of program points that constitute loop entries. The new strategy is defined by

$$\mathcal{S}_{example} \equiv \ p \in \mathcal{P}_{loopentry} \wedge \beta(test(p)) = D \wedge \mathcal{S}_{pointwise}$$

Here, the term $\beta(test(p))$ is a shorthand for stating that the test expression of the loop starting at p is dynamic in β. The above strategy will not always prevent code explosion, and it does not guarantee termination of the specialization phase. However, it demonstrates that reasonable heuristics can be simple to phrase.

An example where this strategy turns out to be useful is shown in Fig. 6. The source program is a fragment of an interpreter for a Fortran-like language with one local and one global scope. Beside the input expression, the position of the global scope in the store is also statically known. However, the store itself and the position of the local scope in the store are dynamic.

The reader may convince himself that a uniform BTA will not achieve satisfactory specialization in this example. As demonstrated [8] in a similar case, the return value of `eval` will be considered dynamic, disallowing full evaluation of $2 + 3$. On the other hand, using a maximally polyvariant BTA, we run into a different problem. In the `WHILE`-loop of procedure `lookup`, there is a possibility

[2] That is, treating both procedure entry and exit fully polyvariantly.

Source code	Residual code of `eval((2+3)+x)`
10: PROCEDURE eval(E): 11: CASE E.op: 12: 'cnst: RETURN E.val; 13: 'var : RETURN lookup(E.id); 14: '+ : RETURN eval(E.Lexp)+ eval(E.Rexp); ... 20: PROCEDURE lookup(Id): 21: COMMON GlobS,LocS,St; 22: INTEGER Cur; 23: Cur:=LocS; 24: WHILE (St[Cur].id≠Id) DO 25: IF (St[Cur]='end) 26: THEN Cur:=GlobS; 27: ELSE Cur:=Cur+1; 28: ENDWHILE 29: RETURN St[Cur].val; 30: END;	(* E = (2+3)+x *) (* GlobS = 0 *) (* LocS and St are dynamic. *) ... 100: COMMON LocS,St; 101: INTEGER Cur; 102: Cur:=LocS; 103: WHILE (St[Cur].id≠'x) DO 104: IF (St[Cur]='end) 105: THEN Cur:=0; 106: ELSE Cur:=Cur+1; 107: ENDWHILE 108: RETURN 5+St[Cur].val;

Fig. 6. Specialization of an interpreter fragment using the example strategy

of variable `Cur` turning static (by assigning to it the value of `GlobS`). This possibility will be explored by the specializer. However, since `Cur` increases under dynamic control, specialization will run into an infinite loop.

Now consider our example strategy. Because of the polyvariant procedure annotation, $(2+3)$ can be completely evaluated. Since the (dynamically controlled) `WHILE`-loop must be annotated monovariantly, `Cur` will always be considered dynamic and we avoid infinite specialization. Thus, we avoid both problems and obtain useful residual code.

8 Related Work

Binding-time analysis for partial evaluation was first developed in the context of functional languages [20] and was later carried over to imperative languages [16, 1]. Analyses for imperative languages are usually more complex due to the different storage model, side-effects, aliasing of variables and pointer manipulations [2, 17]. Binding-time analyses for object-oriented languages are still under development. A multi-level binding-time analysis [15] analyses source programs over an abstract domain representing two or more stages of computation.

Regardless of the source language or the type of analysis, all existing offline partial evaluators fix one particular binding-time strategy for program specialization (e.g., [3, 9, 12, 22]). The most popular strategy, due to its conceptual simplicity, it to annotate programs using a uniform binding-time analysis [19]. The use of a flow-sensitive binding-time analysis for partial evaluation was pioneered in Tempo [17, 18].

Few attempts have been made to examine the impact of different annotation strategies on the quality of the residual programs and the specialization process. Notable exceptions are [11, 6] who developed a polyvariant BTA for a higher-order applicative language, and [23] who implemented a polyvariant BTA for the Similix partial evaluator. An alternative approach was suggested in [8] where polyvariance is achieved by instrumenting programs with explicit bt-values and performing partial evaluation in two passes; [24] used the interpretive approach to the same effect. These works deal with higher-order functional languages.

Strategies for guaranteeing termination of the specialization process without being overly conservative are a topics of current research [5, 14]. These strategies ensure termination by controlling the degree of polyvariance at specialization time by dynamizing appropriate variables. A speed-up analysis that predicts the relative speedup of residual programs obtained using a uniform annotation was studied in [4].

9 Conclusion

Our goal was to develop the foundations for an adaptive approach to binding-time analysis which is flexible and powerful enough to study the impact of binding-time strategies in a realistic context. We advocate that partial evaluation systems be built that allow flexibility in the BTA instead of hard-coding a single strategy on pragmatic grounds. We showed that different BTA strategies drastically influence of the quality of generated programs. The strategy language we developed allows us to catalog and design different BTA strategies.

References

1. L. O. Andersen. C program specialization (revised version). DIKU Report 92/14, DIKU, University of Copenhagen, 1992.
2. L. O. Andersen. Binding-time analysis and the taming of C pointers. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, 1993.
3. L. O. Andersen. Program analysis and specialization for the C programming language. DIKU Report 94/19, Department of Computer Science, University of Copenhagen, 1994.
4. L. O. Andersen and C. K. Gomard. Speedup analysis in partial evaluation: preliminary results. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 1–7, San Francisco, California, 1992. Yale University, Dept. of Computer Science.
5. P. H. Andersen and C. K. Holst. Termination analysis for offline partial evaluation of a higher order functional language. In R. Cousot and D. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 67–82, Aachen, Germany, 1996. Springer-Verlag.
6. J. M. Ashley and C. Consel. Fixpoint computation for polyvariant static analyses of higher-order applicative programs. *ACM TOPLAS*, 16(5):1431–1448, 1994.

7. M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.
8. M. A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 59–65, Copenhagen, Denmark, 1993. ACM Press.
9. M. A. Bulyonkov and D. V. Kochetov. Practical aspects of specialization of Algol-like programs. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Proceedings*, volume 1110 of *Lecture Notes in Computer Science*, pages 17–32, Dagstuhl Castle, Germany, 1996. Springer-Verlag.
10. N. H. Christensen and S. Laursen. Partial evaluation of an imperative language. DIKU Student Report 98-7-3, DIKU, Dept. of Computer Science, University of Copenhagen, 1998.
11. C. Consel. Polyvariant binding-time analysis for applicative languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based program Manipulation*, pages 66–77. ACM Press, 1993.
12. C. Consel and F. Nöel. A general approach for run-time specialization and its application to C. In *Conference Record of the Twenty Third Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 1996. ACM Press.
13. A. P. Ershov and V. E. Itkin. Correctness of mixed computation in Algol-like programs. In J. Gruska, editor, *Mathematical Foundations of Computer Science 1977*, volume 53 of *Lecture Notes in Computer Science*, pages 59–77, Tatranská Lomnica, 1977. Springer-Verlag.
14. A. J. Glenstrup and N. D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 273–284, Novosibirsk, Russia, 1996. Springer-Verlag.
15. R. Glück and J. Jørgensen. Fast binding-time analysis for multi-level specialization. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272, Novosibirsk, Russia, 1996. Springer-Verlag.
16. C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12:123–144, 1991.
17. L. Hornof, C. Consel, and J. Noyé. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Static Analysis. Proceedings*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, 1997. Springer-Verlag.
18. L. Hornof and J. Noyé. Accurate binding-time analysis for imperative languages: flow, context, and return sensitivity. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 63–73, Amsterdam, The Netherlands, 1997. ACM Press.
19. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
20. N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

21. N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.
22. P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. *SIGPLAN Notices*, 30(4):61–70, 1995.
23. B. Rytz and M. Gengler. A polyvariant binding time analysis. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–28, San Francisco, California, 1992. Yale University, Dept. of Computer Science.
24. P. Thiemann and M. Sperber. Polyvariant expansion and compiler generators. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181, pages 285–296, Novosibirsk, Russia, 1996. Springer-Verlag.