Polyvariant Expansion and Compiler Generators

Peter Thiemann and Michael Sperber

Wilhelm-Schickard-Institut, Universität Tübingen
Sand 13, D-72076 Tübingen, Germany
{thiemann,sperber}@informatik.uni-tuebingen.de

Abstract. Polyvariant expansion is a binding-time-improving transformation for offline partial evaluation. We show how to achieve it automatically for a higher-order functional language using the interpretive approach. We have designed and implemented an interpreter that statically propagates binding times. When specialized with respect to a source program, it performs polyvariant expansion. Extending the interpreter to an online specializer allows us to generate a binding-time-polyvariant compiler generator. Maintaining the binding times affords an abstract interpretation equivalent to a constraint-based binding-time analysis.

Keywords: partial evaluation, automatic program transformation, abstract interpretation, program analysis

Partial evaluation is a program specialization technique based on aggressive constant propagation. Given the *static* (known) parameters of a program, partial evaluation constructs a *residual program*—a specialized version of the program, which on application to the remaining *dynamic* parameters produces the same result as the original program applied to all parameters. Offline partial evaluation [7, 22] stages the specialization in two phases, *binding-time analysis* (BTA) and static reduction. The BTA starts with a program and the binding times (static/dynamic) of the parameters and propagates the binding times through the program, producing an annotated program.

Binding-time analyses come in two flavors: A *monovariant* BTA computes a single mapping of program points to binding-times, whereas a *polyvariant* BTA allows for several such mappings. Both alternatives have their merits. Monovariant BTAs are simple and efficient to implement [17, 18, 19, 22, 2], but usually perform poorly in the real world. Quite often, static and dynamic values flow through the same program points, which forces the binding-time analysis to annotate the program points as dynamic. A polyvariant BTA [4, 5] yields better results in real-world programs, but is also considerably more expensive.

When only a monovariant binding-time analysis is available, the programmer usually needs to perform *binding-time improvements* to improve binding-time separation in the subject program. One such binding-time improvement is to duplicate procedures the calls of which appear in multiple binding-time contexts, and changing each call such that it always refers to the appropriate version. Binding-time-monovariant partial evaluation of the binding-time-improved subject program then yields a result equivalent to true polyvariance. This technique is called *polvariant expansion*. Figure 1 shows an example program: The version on the left side calls the procedure `my-map` in three different binding-time contexts. Hence, a monovariant binding-time analysis would force all relevant binding times to become dynamic. Polyvariant expansion results in the code on the right-hand side.

We show how to automate polyvariant expansion for higher-order recursion equations by partial evaluation itself, using the interpretive approach [16]. We apply a binding-time-monovariant partial evaluator to a non-standard interpreter which propagates binding times on-the-fly. Polyvariant specialization (note this is distinct from binding-time polyvariance) of the interpreter with respect to a program and the binding times of its parameters results in a copy of the program with certain parts duplicated, according to the different contexts of use. The specialized program is then amenable to binding-time-monovariant offline partial evaluation.

To avoid reanalyzing the specialized program it is possible to write the interpreter as an online specializer. Specializing the latter with respect to a program and the binding times of the parameters yields a *generating extension* for the program (which accepts the static parameters and computes the residual program). The Futamura projections even lead to the automatic generation of a binding-time-polyvariant compiler generator (or *cogen*—a generator of generating extensions) at the (rather low) cost of writing the interpreter [13, 29, 11, 12]. The construction of a binding-time polyvariant cogen is new.

The following section is an introduction to the interpretive approach and its application to the polyvariant expansion problem. The next section defines the language of the subject programs. Section 3 defines a binding-time domain and its representation as used by our interpreter. Section 4 describes a binding-time analysis which operates on that domain which is the prerequisite for the binding-time-propagating interpreter in the

```
;; main: D × S × D → D
;; Original program
(define (main g xs ys)
  (list (my-map (lambda (y) y) xs)
        (my-map g xs)
        (my-map g ys)))

(define (my-map f xs)
  (if (null? xs)
      '()
      (cons (f (car xs))
            (my-map f (cdr xs)))))
```

```
;; After polyvariant expansion
(define (main g xs ys)
  (list (my-map-1 (lambda (y) y) xs)
        (my-map-2 g xs)
        (my-map-2 g ys)))

;; (S → S) × S → S
(define (my-map-1 f xs)
  (if (null? xs)
      '()
      (cons (f (car xs))
            (my-map-2 f (cdr xs)))))

;; D × S → D
(define (my-map-2 f xs)
  (if (null? xs)
      '()
      (cons (f (car xs))
            (my-map-2 f (cdr xs)))))


;; D × D → D
(define (my-map-3 f xs)
  (if (null? xs)
      '()
      (cons (f (car xs))
            (my-map-3 f (cdr xs)))))
```

Fig. 1. An example for polyvariant expansion

next section. Section 6 is dedicated to the construction of the binding-time polyvariant cogen. Finally, Sec. 7 contains an overview of related work.

1 Using the Interpretive Approach for Polyvariant Expansion

In addition to straightforward compilation [13, 22], partial evaluation can perform almost arbitrary program transformations [15, 16]. The essence of this *interpretive approach* is to specialize a subject program indirectly through an interpreter. Consider an application of the first Futamura projection [13] for an interpreter `int` which returns the result of the program execution via $[\![\texttt{int}]\!]\ \texttt{p}\ \texttt{in}$ for a program p and its input `in`:

$$\mathrm{p}' = [\![\mathrm{pe}]\!]\ \texttt{int}\ \texttt{p}$$

p′ is equivalent to p. Now, mix-style partial evaluation assembles residual programs from pieces of the subject programs. Hence, changing the interpreter `int` to non-standard interpretation techniques leads to changes in p′ [15, 16, 27].

We exploit the fact that mix-style partial evaluators perform *polyvariant specialization*—they may generate multiple specializations of a single program point. To this end, the specializer keeps a *specialization cache* which maps so-called program points (also called *specialization points)* along with the static components of their free variables to appropriate specializations. In the residual program, the specializations become separate procedures.

To achieve polyvariant expansion, we use an interpreter which propagates binding-time information in addition to the actual values computed. The binding-time computation happens completely at specialization time, and thus does not show up in the residual programs. However, for a given specialization point, the specializer generates a new specialization for every possible binding-time configuration because it must assume that they lead to different specializations (which they don't). This is exactly the desired effect of polyvariant expansion.

$$
\begin{array}{lll}
 & & K \in \textsf{Constants} \\
E ::= V \mid & & V \in \textsf{Variables} \\
\quad K \mid & & P \in \textsf{ProcedureNames} \\
\quad \texttt{(}O\ E^*\texttt{)} \mid \texttt{(}P\ E^*\texttt{)} \mid & & O \in \textsf{Operators} \\
\quad \texttt{(}\ell : C\ E^*\texttt{)} \mid \texttt{(}S^C\ E^*\texttt{)} \mid & & T \in \textsf{ConstructorFamilies} \\
\quad \texttt{(if}\ E\ E\ E\texttt{)} \mid & & C \in \textsf{Constructors} \\
\quad \texttt{(}\ell : \texttt{lambda (}V\texttt{)}\ E\texttt{)} \mid \texttt{(}E\ E\texttt{)} & & S^C \in \textsf{Selectors} \\
D ::= \texttt{(define (}P\ V^*\texttt{)}\ E\texttt{)} \mid & & \ell \in \textsf{Labels} \\
\quad \texttt{(define-data}\ T\ \texttt{(}C\ S^{C^*}\texttt{)}^*\texttt{)} & & E \in \textsf{Expressions} \\
\Pi ::= D^+ & & D \in \textsf{Definitions} \\
 & & \Pi \in \textsf{Programs}
\end{array}
$$

Fig. 2. Syntax

2 Source Language

Figure 2 shows the syntax for our subject language—a variant of higher-order recursion equations in Scheme [20] syntax. There are two notable additions: First, there is support for user-defined constructor families, as is common in offline partial evaluation [6, 2]. A declaration

$$\texttt{(define-data}\ T\ \texttt{(}C_1\ S_1^{C_1} \ldots S_{k_1}^{C_1}\texttt{)} \ldots \texttt{(}C_n\ S_1^{C_n} \ldots S_{k_n}^{C_n}\texttt{))}$$

defines a new constructor family T with k_i-ary constructors C_i and selectors $S_j^{C_i}$ for the corresponding fields. In addition, `lambda` abstractions and constructor applications each have a unique label ℓ. For a given subject program, a function $\psi(\ell)$ yields the corresponding `lambda` expression in the program. Also, a function FV : Expressions $\rightarrow$ Variables* maps expressions to ordered sequences of their free variables.

3 Representing Binding Times

The basis for our work is the formulation of a binding-time domain appropriate for use in the interpreter. In this section, we first introduce the domain formally, then show how to restrict it to a finite lattice which leads to a representation amenable to implementation in our interpreter.

3.1 A Binding-Time Domain

The domain of binding times contains a least element $\perp$ for unreachable code, a greatest element $\top$ for dynamic computations, and type constructors C applied to zero or more arguments. Among the type constructors, B denotes base values. The remaining constructors are specific to the subject program: Each occurrence of a `lambda` at program point ℓ has its own type constructor clo_ℓ where the arguments are the binding times of the free variables of the `lambda`. Furthermore, formal disjunctions of binding times are possible.

$$b ::= \perp \mid \top \mid C\ b \ldots b \mid b \vee b$$

$\vee$ is a commutative and associative operation with unit $\perp$ and $b \vee \top = \top$ which is compatible with type constructors $(C\ b_1 \ldots b_n \vee C\ b'_1 \ldots b'_n = C(b_1 \vee b'_1) \ldots (b_n \vee b'_n))$. This induces an infinite lattice. Each element is either $\perp$, $\top$, or a finite disjunction of constructor applications where all constructors are different. $C\ b_1 \ldots b_n \in b$ means that $b = \ldots C\ b_1 \ldots b_n \ldots$. We also include infinite terms.

3.2 Representing the Binding-Time Lattice

To make our binding-time domain amenable to finite representations, we restrict it to the sublattice of regular terms which possess only finitely many different subterms. (The terms may still be infinite.) From this lattice a finite lattice results by considering only those terms where each subterm is already uniquely identified by its type constructor: For every two occurrences $C\ b_1 \ldots b_n$ and $C\ b'_1 \ldots b'_n$ in a given binding-time value, we require that $b_i = b'_i$, for $1 \leq i \leq n$. Thus, a single binding-time value specifies a set of regular constructor terms.

The concrete representation of binding-time values is a triple $(\ell, \sigma, \sim)$ consisting of a program point ℓ, a set σ of equations of the form $\ell = R$, and an equivalence relation $\sim$ on L, the set of program points. The right-hand sides of equations are described by

$$R ::= \top \mid C\ell \ldots \ell$$

The binding-time value denoted by $(\ell, \sigma, \sim)$ is $\rho(\ell, \sigma, \sim) = C_1\ b_{11} \ldots b_{1n_1} \vee \ldots \vee C_m\ b_{m1} \ldots b_{mn_m}$ if σ contains the pairs $\ell_j = C_j\ \ell_{j1} \ldots \ell_{jn_j}$ where $\{\ell_1, \ldots, \ell_m\}$ is the set of all left sides which are equivalent to ℓ and $\rho(\ell_{ji}, \sigma, \sim) = b_{ji}$. Otherwise $\rho(\ell, \sigma, \sim)$ is $\top$ if σ contains $\ell' = \top$ for $\ell \sim \ell'$, or $\perp$ if nothing else applies.

Normal Form To avoid infinitely growing representations, we define a normal form for representations. In normal form, σ contains at most one equation $\ell = C \ldots$ for each pair $([\ell]_\sim, C)$ (or an equation $\ell = \top$). The following rewriting rules induce an algorithm which transforms an arbitrary representation to normal form. For a fixed source program, we assume a function $\Gamma : \mathsf{Constructors} \to \mathsf{ConstructorFamilies}$ mapping each constructor to the constructor family of its declaration, with $\Gamma clo_\ell = \text{closure}$.

$$\frac{(\ell, \sigma, \sim)}{(\ell, \sigma \cup \{\ell_0 = \top\}, \sim)} \begin{array}{l}\text{if } \sigma \supseteq \{\ell_0 = C\ell_1 \ldots \ell_n, \ell'_0 = C'\ell'_1 \ldots \ell'_m\} \\ \text{and } \ell_0 \sim \ell'_0 \wedge \Gamma C \neq \Gamma C'\end{array} \tag{1}$$

$$\frac{(\ell, \sigma \cup \{\ell_0 = C\ell_1 \ldots \ell_n, \ell'_0 = C\ell'_1 \ldots \ell'_n\}, \sim)}{(\ell, \sigma \cup \{\ell_0 = C\ell_1 \ldots \ell_n\}, (\sim \cup \{(\ell_i, \ell'_i)\})^{\cong})} \text{ if } \ell_0 \sim \ell'_0 \tag{2}$$

(For a binary relation R, $R^{\cong}$ denotes its reflexive, transitive, and symmetric closure.)

$$\frac{(\ell, \sigma \cup \{\ell_0 = C\ell_1 \ldots \ell_n, \ell'_0 = \top\}, \sim)}{(\ell, \sigma \cup \{\ell_0 = \top, \ell_1 = \top, \ldots, \ell_n = \top\}, \sim)} \text{ if } \ell_0 \sim \ell'_0 \wedge \Gamma C \neq \text{closure} \tag{3}$$

Setting the arguments of a constructor to $\top$ is required to match the binding-time analysis in Sec. 4 which is locally monovariant. In contrast, closures require a different rule:

$$\frac{(\ell, \sigma \cup \{\ell_0 = C\ell_1 \ldots \ell_n, \ell'_0 = \top\}, \sim)}{(\ell, \sigma \cup \{\ell_0 = \top\}, \sim)} \text{ if } \Gamma C = \text{closure} \tag{4}$$

Strong Normal Form Even a binding-time representation in normal form can contain superfluous equations. Removing those leads to a stronger formulation of normalization:

An equation $\ell' = R \in \sigma$ is *reachable* in $(\ell, \sigma, \sim)$ if $\ell' \sim \ell$ or there exists $\ell'' = C\ell_1 \ldots \ell_n$ in σ with $\ell'' \sim \ell$ and $\ell' = R$ is reachable in $(\ell_j, \sigma, \sim)$, for some $1 \leq j \leq n$. The last rule allows us to drop unreachable equations from σ:

$$\frac{(\ell, \sigma \cup \{\ell_0 = R\}, \sim)}{(\ell, \sigma, \sim)} \text{ if } \ell_0 = R \text{ is not reachable in } (\ell, \sigma, \sim) \tag{5}$$

A representation is in *strong normal form* if it is in normal form and all of its equations are reachable. Repeated application of the rewriting rules terminates and results in a binding-time value in strong normal form. Let N be an algorithm that exhaustively applies the above rewriting rules.

Operations on Binding Times For brevity's sake, $\top$ is a shorthand for the binding-time representation $(\ell_\top, \ell_\top = \top, =)$, and B stands for $(\ell_B, \ell_B = B, =)$ where $\ell_\top$ and ℓ_B are unique labels.

With the help of N, the least upper bound on binding-time representations is

$$(\ell, \sigma, \sim) \sqcup (\ell', \sigma', \sim') := N(\ell, \sigma \cup \sigma', (\sim \cup \sim' \cup \{(\ell, \ell')\})^{\cong})$$

Two auxiliary operations are necessary to support our interpreter—one for constructor application, and one for selection from a constructed type.

$$\widetilde{C}(\ell, \ldots, (\ell_i, \sigma_i, \sim_i), \ldots) := N(\ell, \{\ell = C\ \ell_1 \ldots \ell_n\} \cup \sigma_1 \cup \ldots \cup \sigma_n, (\sim_1 \cup \ldots \cup \sim_n)^{\cong})$$

Here, ℓ is the program point of the constructor application. Finally, selection of the ith argument of constructor C at point ℓ' is defined by

$$C.i(\ell, \sigma, \sim) = \begin{cases} \top & \text{if } \forall \ell'' \sim \ell. \ell'' = R \in \sigma \Rightarrow R \neq C\ell_1 \ldots \ell_n \\ N(\ell_i, \sigma, \sim) & \text{if } \exists \ell'' \sim \ell. \ell'' = C\ell_1 \ldots \ell_n \in \sigma \end{cases}$$

$$
\begin{aligned}
&\mathcal{B}[\![V]\!]\beta &&= \beta[\![V]\!]\\
&\mathcal{B}[\![K]\!]\beta &&= B\\
&\mathcal{B}[\![(\ell : C\ E_1 \ldots E_n)]\!]\beta &&= \widetilde{C}(\ell, \mathcal{B}[\![E_1]\!]\beta, \ldots, \mathcal{B}[\![E_n]\!]\beta)\\
&\mathcal{B}[\![(S_i^C\ E)]\!] &&= C.i(\mathcal{B}[\![E]\!]\beta)\\
&\mathcal{B}[\![(O\ E_1 \ldots E_n)]\!]\beta &&= \textstyle\bigsqcup \mathcal{B}[\![E_i]\!]\beta\\
&\mathcal{B}[\![(\texttt{if}\ E_1\ E_2\ E_3)]\!]\beta &&= \textstyle\bigsqcup \mathcal{B}[\![E_i]\!]\beta\\
&\mathcal{B}[\![(P\ E_1 \ldots E_n)]\!]\beta &&= \mathcal{B}[\![E_P]\!][V_i \mapsto \mathcal{B}[\![E_i]\!]\beta]\\
&\mathcal{B}[\![(\ell : \texttt{lambda}\ (V)\ E)]\!]\beta &&= let\ V_1' \ldots V_m' = FV(E)\ in\ clo_\ell\ (\beta[\![V_1']\!]) \ldots (\beta[\![V_m']\!])\\
&\mathcal{B}[\![(E_1\ E_2)]\!]\beta &&= let\ b_i = \mathcal{B}[\![E_i]\!]\beta\ in\\
& && \quad if\quad b_1 = \top\ then\ \top\\
& && \quad else \textstyle\bigsqcup\{\mathcal{B}[\![E]\!][V_i' \mapsto b_i', V \mapsto b_2] \mid clo_\ell b_1' \ldots b_m' \in b_1, \psi(\ell) = (\texttt{lambda}\ (V)\ E),\\
& && \qquad\qquad\qquad\qquad\qquad\qquad V_1' \ldots V_m' = FV(E)\}
\end{aligned}
$$

Fig. 3. Binding-time analysis

4 Computing Binding Times

The actual binding-time analysis function shown in Fig. 3 takes an expression and a binding-time environment (mapping variables to binding-time values) and returns a binding-time value.

In order to determine the binding time of a call to a recursive function, a fixpoint must be computed. We omit the standard extension of $\mathcal{B}[\![\_]\!]$ with a cache which ensures termination of the computation [5].

5 The Interpreter

As the interpreter must keep track of both binding times and values, we need two environments β and ρ. β maps variables to binding times and ρ to their values. We also need a stack γ of binding times with empty stack $\diamond$, and the binary operation : denoting a push (from the left). The operations *pop* and *top* are defined as usual. We also require $pop\ \diamond = \diamond$ and $top\ \diamond = \top$ from the binding-time domain. The function $\mathcal{K}[\![\_]\!]$ maps constants K to values, $\mathcal{O}[\![\_]\!]$ maps built-in operators O to functions, and $\mathcal{C}[\![\_]\!]$ maps constructors and selectors to corresponding built-in functions. The metalanguage is an enriched call-by-value lambda calculus with $\_ \rightarrow \_, \_$ denoting the McCarthy conditional. $\mathcal{B}[\![E]\!]\beta$ computes the binding time of E in the binding-time environment β; it is defined in Fig. 3.

$$
\begin{aligned}
&\mathcal{N}[\![V]\!]\beta\gamma\rho &&= \rho[\![V]\!]\\
&\mathcal{N}[\![K]\!]\beta\gamma\rho &&= \mathcal{K}[\![K]\!]\\
&\mathcal{N}[\![(O\ E_1 \ldots E_n)]\!]\beta\gamma\rho &&= \mathcal{O}[\![O]\!](\mathcal{N}[\![E_1]\!]\beta\diamond\rho \ldots \mathcal{N}[\![E_n]\!]\beta\diamond\rho)\\
&\mathcal{N}[\![(C\ E_1 \ldots E_n)]\!]\beta\gamma\rho &&= \mathcal{C}[\![C]\!](\mathcal{N}[\![E_1]\!]\beta\diamond\rho \ldots \mathcal{N}[\![E_n]\!]\beta\diamond\rho)\\
&\mathcal{N}[\![(S_i^C\ E)]\!]\beta\gamma\rho &&= \mathcal{C}[\![S_i^C]\!](\mathcal{N}[\![E]\!]\beta\diamond\rho)\\
&\mathcal{N}[\![(\texttt{if}\ E_1\ E_2\ E_3)]\!]\beta\gamma\rho &&= \mathcal{N}[\![E_1]\!]\beta\diamond\rho \rightarrow \mathcal{N}[\![E_2]\!]\beta\gamma\rho, \mathcal{N}[\![E_3]\!]\beta\gamma\rho\\
&\mathcal{N}[\![(P\ E_1 \ldots E_n)]\!]\beta\gamma\rho &&= \mathcal{N}[\![E_P]\!][V_i \mapsto \mathcal{B}[\![E_i]\!]\beta][V_i \mapsto \mathcal{N}[\![E_i]\!]\beta\diamond\rho]\\
&\mathcal{N}[\![(\texttt{lambda}\ (V)\ E)]\!]\beta\gamma\rho &&= \lambda y.\mathcal{N}[\![E]\!]\beta[V \mapsto top\ \gamma](pop\ \gamma)\rho[V \mapsto y]\\
&\mathcal{N}[\![(E_1\ E_2)]\!]\beta\gamma\rho &&= (\mathcal{N}[\![E_1]\!]\beta(\mathcal{B}[\![E_2]\!]\beta : \gamma)\rho)(\mathcal{N}[\![E_2]\!]\beta\diamond\rho)
\end{aligned}
$$

Fig. 4. Interpreter with binding-time propagation

Figure 4 shows the non-standard interpreter. The interesting aspects are the computation of binding times for the parameters of procedure calls, and the handling of the binding-time stack γ. This stack is necessary to provide binding times for the arguments of `lambda`s. Each application that the interpreter encounters pushes the binding time of its argument when descending into the function part of the evaluation. At the corresponding `lambda`, the interpreter retrieves the binding time of the argument from the top of the stack. It continues the interpretation of the body of `lambda` with the popped stack.

Binding times in the interpreter are clearly separated if β and γ are seeded from static data: β, γ, and the program are static; ρ is dynamic. However, in the interpreter shown in Fig. 4, the stack γ may grow without bounds, leading to nonterminating specialization. In our actual implementation, the remedy is a generalization algorithm which confines the stack to finite variation. Whenever the implementation recognizes a loop on the stack, it fold the loop back onto its previous iteration and applies a least upper bound operation. Finiteness of the program and the binding-time domain ensures finiteness of this process.

6 Constructing a Cogen

$$
\begin{array}{ll}
\mathcal{S}[\![V]\!]\beta\gamma\rho & = \rho[\![V]\!] \\
\mathcal{S}[\![K]\!]\beta\gamma\rho & = \mathit{base}(\mathcal{K}[\![K]\!]) \\
\mathcal{S}[\![(O\ E_1 \ldots E_n)]\!]\beta\gamma\rho & = (\mathcal{B}[\![(O\ E_1 \ldots E_n)]\!]\beta = B) \rightarrow \\
& \quad \mathcal{O}[\![O]\!](\mathcal{S}[\![E_1]\!]\beta\Diamond\rho, \ldots, \mathcal{S}[\![E_n]\!]\beta\Diamond\rho), \\
& \quad \mathit{build\text{-}O}(\mathit{lift}\ (\mathcal{B}[\![E_1]\!]\beta, \mathcal{S}[\![E_1]\!]\beta\Diamond\rho), \ldots, \mathit{lift}\ (\mathcal{B}[\![E_n]\!]\beta, \mathcal{S}[\![E_n]\!]\beta\Diamond\rho)) \\
\mathcal{S}[\![(\ell : C\ E_1 \ldots E_n)]\!]\beta\gamma\rho & = \mathit{construction}(C, \mathcal{S}[\![E_1]\!]\beta\Diamond\rho \ldots \mathcal{S}[\![E_n]\!]\beta\Diamond\rho) \\
\mathcal{S}[\![(S_i^C\ E)]\!]\beta\gamma\rho & = (\mathcal{B}[\![E]\!]\beta \neq \top) \rightarrow \\
& \quad (\mathit{let}\ \mathit{construction}(C, f_1 \ldots f_n) = \mathcal{S}[\![E]\!]\beta\Diamond\rho \\
& \quad\ \ \mathit{in}\ f_i), \\
& \quad (\mathit{build\text{-}S_i^C}(\mathit{lift}(\mathcal{B}[\![(S_i^C\ E)]\!]\beta, \mathcal{S}[\![E]\!]\beta\Diamond\rho))) \\
\mathcal{S}[\![(\texttt{if}\ E_1\ E_2\ E_3)]\!]\beta\gamma\rho & = (\mathcal{B}[\![E_1]\!]\beta = B) \rightarrow \\
& \quad (\mathcal{S}[\![E_1]\!]\beta\gamma\rho \rightarrow \mathcal{S}[\![E_2]\!]\beta\gamma\rho, \mathcal{S}[\![E_3]\!]\beta\gamma\rho), \\
& \quad (\mathit{build\text{-}if}(\mathit{lift}(\mathcal{B}[\![E_1]\!]\beta, \mathcal{S}[\![E_1]\!]\beta\gamma\rho), \mathit{lift}(\mathcal{B}[\![E_2]\!]\beta, \mathcal{S}[\![E_2]\!]\beta\gamma\rho), \mathit{lift}(\mathcal{B}[\![E_3]\!]\beta, \mathcal{S}[\![E_3]\!]\beta\gamma\rho))) \\
\mathcal{S}[\![(P\ E_1 \ldots E_n)]\!]\beta\gamma\rho & = \mathcal{S}[\![E_P]\!][V_i \mapsto \mathcal{B}[\![E_i]\!]\beta]\gamma[V_i \mapsto \mathcal{S}[\![E_i]\!]\beta\Diamond\rho] \\
\mathcal{S}[\![(\ell : \texttt{lambda}\ (V)\ E)]\!]\beta\gamma\rho & = \mathit{let}\ V_1 \ldots V_n = FV((\texttt{lambda}\ (V)\ E)) \\
& \qquad\quad f_1 \ldots f_n = \rho[\![V_1]\!] \ldots \rho[\![V_n]\!] \\
& \qquad\quad b_1 \ldots b_n = \beta[\![V_1]\!] \ldots \beta[\![V_n]\!] \\
& \qquad\qquad\quad \beta' = [V_1 \mapsto b_1, \ldots, V_n \mapsto b_n] \\
& \qquad\qquad\quad \gamma' = \mathit{pop}\ \gamma \\
& \quad \mathit{in}\ \ \mathit{closure}(\lambda f_1' \ldots f_n'. \\
& \qquad\qquad\qquad \mathit{build\text{-}lambda}(V^\diamond, \\
& \qquad\qquad\qquad\qquad \mathit{lift}(\mathcal{B}[\![E]\!]\beta'[V \rightarrow \top], \\
& \qquad\qquad\qquad\qquad\quad \mathcal{S}[\![E]\!]\beta'[V \rightarrow \top]\gamma'[V_1 \mapsto f_1', \ldots, V_n \mapsto f_n', V \mapsto V^\diamond])), \\
& \qquad\qquad \lambda y.\lambda f_1' \ldots f_n'.\mathcal{S}[\![E]\!]\beta'[V' \mapsto \mathit{top}]\gamma'[V_1 \mapsto f_1', \ldots, V_n \mapsto f_n', V \mapsto y], \\
& \qquad\qquad f_1 \ldots f_n) \\
\mathcal{S}[\![(E_1\ E_2)]\!]\beta\gamma\rho & = \mathit{let}\ b_1 = \mathcal{B}[\![E_1]\!]\beta \\
& \qquad\quad b_2 = \mathcal{B}[\![E_2]\!]\beta \\
& \quad \mathit{in}\ (b_1 \neq \top) \rightarrow \\
& \qquad (\mathit{let}\ \mathit{closure}(\mathcal{R}, \mathcal{A}, f_1 \ldots f_n) = \mathcal{S}[\![E_1]\!]\beta(b_2 : \gamma)\rho \\
& \qquad\ \ \mathit{in}\ \mathcal{A}(\mathcal{S}[\![E_2]\!]\beta\Diamond\rho, f_1 \ldots f_n)), \\
& \qquad (\mathit{build\text{-}application}(\mathit{lift}(b_1, \mathcal{S}[\![E_1]\!]\beta(b_2 : \gamma)\rho), \mathit{lift}(b_2, \mathcal{S}[\![E_2]\!]\beta\Diamond\rho)))
\end{array}
$$

$$
\begin{array}{ll}
\mathit{lift}(\mathit{base}(K)) & = \mathit{build\text{-}constant}(K) \\
\mathit{lift}(\mathit{construction}(C, f_1 \ldots f_n)) & = \mathit{build\text{-}C}(f_1, \ldots, f_n) \\
\mathit{lift}(\mathit{closure}(\mathcal{R}, \mathcal{A}, f_1 \ldots f_n)) & = \mathcal{R}(f_1 \ldots f_n)
\end{array}
$$

Fig. 5. Polyvariant specializer

A cogen can be constructed by changing the interpreter $\mathcal{N}[\![\_]\!]$ to a specializer $\mathcal{S}[\![\_]\!]$, driven by the propagated binding times. Figure 5 shows a specification of the specializer, albeit omitting a memoization mechanism for the sake of brevity. Note that the specialization of `if` also needs to coerce both branches to equal binding times—a technicality omitted in the specification.

In the specification, *build-X* builds a residual expression corresponding to operator X. Specialization-time values are either constants $\mathit{base}(K)$, constructed data $\mathit{construction}(C, f_1 \ldots f_n)$ with constructor C and components $f_1, \ldots, f_n$, or closures $\mathit{closure}(\mathcal{R}, \mathcal{A}, f_1 \ldots f_n)$, where component $\mathcal{R}$ is a function that performs residualization, $\mathcal{A}$ performs application, and $f_1, \ldots, f_n$ are the values of the free variables of the closure. $V^\diamond$ is a freshly generated identifier. The operation $\mathit{lift}(b, x)$ builds a coercion of a value x of binding time b to dynamic. Specialization of $\mathcal{S}[\![\_]\!]$ with respect to some program p and the initial binding times yields a generating extension for the polyvariant expansion of p. Application of a cogen to $\mathcal{S}[\![\_]\!]$ yields a polyvariant-expanding cogen.

7 Related Work

The contribution of this paper extends prior work in several directions. Several authors have proposed polyvariant expansion to augment binding-time-monovariant partial evaluation. Gengler and Rytz [14, 26] show how to perform polyvariant expansion for higher-order programs by iterating an analysis of the source program that creates new versions in each iteration until no further expansion is necessary. The termination of their method [14, 26] relies on the finiteness of the rather coarse abstract domain of the BTA of an early version of Similix [1]. Our analysis uses an expressive abstract domain which is amenable to a variant of constraint-based BTA [19, 2] and works without the need for multiple iterations of the entire binding-time analysis.

Bulyonkov [3] uses specialization for polyvariant expansion. However, his source language is only first-order and his method is not completely automatic. We have identified his method as an application of the interpretive approach.

Our abstract domain is similar to one used by Mogensen [23]. There is also a close relation to grammar-based analyses [21, 24, 28]. Moreover, mapping an arbitrary b to the least b' in the restricted lattice can be regarded as a widening operator [8, 9, 10].

The idea of using self-application to obtain stand-alone generating extensions came from Futamura [13]. Turchin and Ershov later extended the idea to compiler generators [29, 11, 12].

8 Conclusion

We have implemented polyvariant expansion for higher-order functional programs using the interpretive approach. Our method uses a fine-grained abstract binding-time domain. It fully automates Bulyonkov's method and extends it to higher-order programs, and we have implemented it. In addition, our method can produce in a polyvariant compiler generator from a monovariant partial evaluator, avoiding an additional binding-time analysis of the resulting program. Therefore, our approach provides a simple and practical method for achieving binding-time-polyvariant partial evaluation.

References

1. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.
2. A. Bondorf and J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
3. M. A. Bulyonkov. Extracting polyvariant binding times from polyvariant specializer. In PEPM1993 [25], pages 59–65.
4. C. Consel. Binding time analysis for higher order untyped functional languages. In *Symp. Lisp and Functional Programming '92*, pages 264–272, San Francisco, Ca., June 1992. ACM.
5. C. Consel. Polyvariant binding-time analysis for applicative lanuages. In PEPM1993 [25], pages 66–77.
6. C. Consel. A tour of Schism. In PEPM1993 [25], pages 134–154.
7. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Symposium on Principles of Programming Languages '93*, pages 493–501, Charleston, Jan. 1993. ACM.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th Symposium on Principles of Programming Languages*. ACM, 1977.
9. P. Cousot and R. Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proc. Programming Language Implementation and Logic Programming '92*, pages 269–295, Leuven, Belgium, Aug. 1992. Springer-Verlag. LNCS 631.
10. P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In S. Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 170–181, La Jolla, CA, June 1995. ACM Press, New York.
11. A. Ershov. Mixed computation: Potential applications and problems for study. In *Mathematical Logic Methods in AI Problems and Systematic Programming, Part 1*, pages 26–55. Vil'nyus, USSR, 1980. (In Russian).
12. A. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18:41–67, 1982.
13. Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
14. M. Gengler and B. Rytz. A polyvariant binding time analysis handling partially known values. In *Workshop on Static Analysis*, volume 81–82 of *Bigre Journal*, pages 322–330, Rennes, France, 1992. IRISA.

15. R. Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, Oct. 1994.
16. R. Glück and J. Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, pages 183–194. IEEE Computer Society Press, 1994.
17. C. K. Gomard. Partial type inference for untyped functional programs. In *Proceedings of the Conference on Lisp and Functional Programming*, pages 282–287, Nice, France, 1990. ACM.
18. C. K. Gomard and N. D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.
19. F. Henglein. Efficient type inference for higher-order binding-time analysis. In *Conf. Functional Programming Languages and Computer Architecture '91*, pages 448–472, Cambridge, Sept. 1991. ACM.
20. IEEE. Standard for the Scheme programming language. Technical Report 1178-1990, Institute of Electrical and Electronic Engineers, Inc., New York, 1991.
21. N. D. Jones. *Flow Analysis of Lazy Higher-Order Functional Programs*, pages 103–122. Ellis Horwood, 1987.
22. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
23. T. Æ. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Díaz and F. Orejas, editors, *TAPSOFT '89*, pages II, 298–312, Barcelona, Spain, Mar. 1989. Springer-Verlag. LNCS 351,352.
24. T. Æ. Mogensen. Separating binding times in language specifications. In *Proc. Functional Programming Languages and Computer Architecture 1989*, pages 14–25, London, GB, 1989.
25. *Proc. 1993 ACM Symp. Partial Evaluation and Semantics-Based Program Manipulation*, Copenhagen, Denmark, June 1993. ACM.
26. B. Rytz and M. Gengler. A polyvariant binding time analysis. In C. Consel, editor, *Workshop Partial Evaluation and Semantics-Based Program Manipulation '92*, pages 21–28, San Francisco, CA, June 1992. Yale University. Report YALEU/DCS/RR-909.
27. M. Sperber, R. Glück, and P. Thiemann. Bootstrapping higher-order program transformers from interpreters. In *1996 ACM Symposium on Applied Computing Programming Languages Track*, pages 408–413, Philadelphia, 1996.
28. M. H. Sørensen. A grammar-based data-flow analysis to stop deforestation. In *Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, Edinburgh, Apr. 1994.
29. V. Turchin. A supercompiler system based on the language Refal. *SIGPLAN Notices*, 14(2):46–54, February 1979.

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