**Basic Research in Computer Science**

# Formal Aspects of Polyvariant Specialization

**Henning Korsholm Rohde**

**BRICS Report Series** **RS-05-34**

**ISSN 0909-0878** **November 2005**

**See back inner page for a list of recent BRICS Report Series publications.**
**Copies may be obtained by contacting:**

**BRICS**
**Department of Computer Science**
**University of Aarhus**
**Ny Munkegade, building 540**
**DK–8000 Aarhus C**
**Denmark**
**Telephone: +45 8942 3360**
**Telefax: +45 8942 3255**
**Internet: BRICS@brics.dk**

**BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:**

`http://www.brics.dk`
`ftp://ftp.brics.dk`
**This document in subdirectory** `RS/05/34/`

# Formal Aspects of Polyvariant Specialization

Henning Korsholm Rohde
BRICS*, Department of Computer Science
University of Aarhus, Denmark
hense@brics.dk

November 7, 2005

**Abstract**

We present the first formal correctness proof of an offline polyvariant specialization algorithm for first-order recursive equations. As a corollary, we show that the specialization algorithm generates a program implementing the search phase of the Knuth-Morris-Pratt algorithm from an inefficient but binding-time-improved string matcher.

*Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

# Contents

# 1 Introduction

Partial evaluation is a program-transformation technique characterized by constant propagation and call unfolding [13, 33]. Given a program `f(x,y)=e` and a value $c$ for `x`, a partial evaluator (also called a program specializer) generates a program, $\texttt{f}_c\texttt{(y)=e'}$, equivalent to `f'(y)=f(c,y)`. The usefulness of partial evaluation is that $\texttt{f}_c\texttt{(y)=e'}$ may, due to optimizations enabled by fixing `x`$= c$, be significantly more efficient than `f'(y)=f(c,y)`.

Consider the power function as an example. (It will also be used to illustrate some technical aspects about the main proof in Section 2.4.2.)

```
power(n,x) = if (n=0) then 1 else x*(power(n-1,x))
```

Fixing $n = 2$, a partial evaluator generates:

$$\texttt{power}_2\texttt{(x) = x*(x*1)}$$

In offline partial evaluation, the decision on what can and should be reduced is taken during a so-called binding-time-analysis prephase, and represented as annotations in the program. These annotations are then blindly followed in the subsequent specialization phase. The utility of this division was demonstrated by Jones's group [34] in connection with compiler-generation by self-application of the specializer, an idea originally conceived by Futamura [24].

The above example is annotated as follows (with reducible terms in italics)

`power(`$n$`,x) = `*if (n=0) then* `1` *else* `x*(`*power(n−1,*`x`*)*`)`

Note that it can be annotated before the value of $n$ is chosen. Fixing $n = 2$, the result of specializing the annotated program is as above.

Polyvariant specialization allows recursive functions to be indexed by their fixed arguments, and was originally conceived in Ershov's group [11].

In the example, we may thus choose to index the recursive calls to `power` by $n$, instead of symbolically unfolding them:

$$\begin{aligned}&\texttt{power}_2\texttt{(x) = x*(power}_1\texttt{(x))}\\&\texttt{power}_1\texttt{(x) = x*(power}_0\texttt{(x))}\\&\texttt{power}_0\texttt{(x) = 1}\end{aligned}$$

Here, symbolic unfolding is probably to be preferred to remove the overhead of function calls. To see where polyvariance makes a difference, consider the following mutually-recursive functions:

```
even(n,x) = if (x=0) then true  else odd (n+1,x-1)
odd (n,x) = if (x=0) then false else even(n−1,x-1)
```

Fixing $n = 2$ for even, a monovariant specializer using symbolic unfolding would diverge, because each function unfolds to itself with only the dynamic arguments changed. To avoid non-termination, weaker binding-time annotations must be chosen (i.e., where all terms are annotated as dynamic); the result is trivial specialization:

```
even2(x)  = even(2,x)
even(n,x) = if (x=0) then true  else odd (n+1,x-1)
odd (n,x) = if (x=0) then false else even(n-1,x-1)
```

In contrast, a polyvariant specializer generates:

$$\begin{aligned}&\texttt{even}_2\texttt{(x) = if (x=0) then true  else odd}_3\texttt{(x-1)}\\&\texttt{odd}_3\texttt{(x)  = if (x=0) then false else even}_2\texttt{(x-1)}\end{aligned}$$

As a footnote, the tension between ensuring termination while avoiding trivial specialization is a major issue in partial evaluation [32].

Although the above example is short, the classical example where polyvariance makes a difference is the generation of the search phase of the efficient Knuth-Morris-Pratt algorithm [35] from an inefficient string matcher [12, 26].

### Outline

This work presents the first formal correctness proof of an offline polyvariant specialization algorithm for typed first-order recursive equations. We work in a call-by-value denotational setting [43], where our approach – inspired by type-directed partial evaluation [14, 19, 21] – is to phrase polyvariant specialization as a non-standard, code-generating semantics (also called a residualizing semantics). This approach immediately suggests certain proof principles, namely structural induction (by compositionality) and fixed-point induction.

A priori, two issues complicate the proof:

1. We must keep track of the polyvariant element, i.e., the transitive closure of needed specialized functions. This turns out to be relatively unproblematic and isolated.
2. We must ensure that the result of symbolic unfolding is well-typed and with the right meaning. A well-known complication is that symbolic unfolding necessitates $\alpha$-renaming of let-expressions to avoid variable captures. Symbolic unfolding also turns out to destroy the obvious proof approach in the denotational setting.

As an application, the specialization algorithm is shown to generate a program implementing the search phase of the Knuth-Morris-Pratt algorithm from an inefficient, but binding-time-improved, string matcher.

# 2 Polyvariant specialization

## 2.1 A typed first-order language

We consider first-order recursive equations with a call-by-value denotational semantics [20, 43]. We use standard denotational concepts throughout the paper.

### 2.1.1 Syntax

The language is parameterized by a signature, $\Sigma = (\mathrm{B}, \mathrm{L}, \mathrm{C})$. B is a set of base types, b; L maps base types to disjoint sets of literals, $\mathrm{l} \in \mathrm{L(b)}$; and C maps function constants, c, to function types.

**Types:**

$$\begin{array}{lll} \mathit{GTyp}: & \tau ::= & \mathrm{b} \mid \texttt{bool} \qquad \text{(Ground types)} \\ \mathit{FTyp}: & \sigma ::= & \tau_1 \texttt{*} \ldots \texttt{*} \tau_n \texttt{->} \tau \qquad \text{(Function types)} \end{array}$$

**Terms:** Let $x \in \mathit{Var}$ and $f \in \mathit{FVar}$ denote variables. We define expressions, equations (or functions), declarations, and programs. A notable restriction is that arguments to function calls must be variables. This restriction simplifies the formal construction, because it will eliminate the need for the partial evaluator to insert `let`s to avoid unsound handling of computations [9]. Hence, we effectively require that source programs have already been `let`-inserted (even for literals, but here only to simplify the syntax).

$$\begin{array}{lll} \mathit{Exp}: & e ::= & \mathrm{l} \mid \texttt{true} \mid \texttt{false} \mid \mathrm{c}\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid \\ & & x \mid \texttt{let}\, x\texttt{:}\tau_1 \Leftarrow e_1\, \texttt{in}\, e_2 \mid f\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)} \\ \mathit{Eqn}: & q ::= & f\texttt{:}\sigma\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)=}e \\ \mathit{Decl}: & d ::= & \texttt{[}q_1\texttt{,}\ldots\texttt{,}q_n\texttt{]} \\ \mathit{Pgm}: & p ::= & \texttt{local}\, d\, \texttt{in}\, f \end{array}$$

Equation names are required to be distinct within a declaration and argument names are required to be distinct within each equation.

For any $a, b \in \mathbb{N}$, let $[a..b]$ denote $\{a, a+1, \ldots, b\}$ if $a \leq b$, and $\emptyset$ if $a > b$.

### 2.1.2 Typing

Let $\Gamma : \mathit{Var} \to \mathit{GTyp}$, and $\Psi : \mathit{FVar} \to \mathit{FTyp}$ be typing environments. We define a relation, $\Psi, \Gamma \vdash e : \tau$, asserting that $e$ is a well-typed term of type $\tau$, in the context of $\Psi$ and $\Gamma$. Similarly, $\Psi \vdash q : (f, \sigma)$, $\vdash d : \Psi$, and $\vdash p : \sigma$.

$\boxed{\Psi, \Gamma \vdash e : \tau}$

$$\frac{\mathrm{l} \in \mathrm{L(b)}}{\Psi, \Gamma \vdash \mathrm{l} : \mathrm{b}}$$

$$\frac{}{\Psi, \Gamma \vdash \texttt{true} : \texttt{bool}}$$

$$\frac{}{\Psi, \Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{\mathrm{C(c)} = \tau_1 \texttt{*} \ldots \texttt{*} \tau_n \texttt{->} \tau \quad \forall i \in [1..n].\, \Psi, \Gamma \vdash e_i : \tau_i}{\Psi, \Gamma \vdash \mathrm{c}(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Psi, \Gamma \vdash e_1 : \texttt{bool} \quad \Psi, \Gamma \vdash e_2 : \tau \quad \Psi, \Gamma \vdash e_3 : \tau}{\Psi, \Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Psi, \Gamma \vdash x : \tau}$$

$$\frac{\Psi, \Gamma \vdash e_1 : \tau_1 \quad \Psi, \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Psi, \Gamma \vdash \texttt{let}\, x \texttt{:} \tau_1 \Leftarrow e_1\, \texttt{in}\, e_2 : \tau_2}$$

$$\frac{\Psi(f) = \tau_1 \texttt{*} \ldots \texttt{*} \tau_n \texttt{->} \tau \quad \forall i \in [1..n].\, \Gamma(x_i) = \tau_i}{\Psi, \Gamma \vdash f(x_1, \ldots, x_n) : \tau}$$

$\boxed{\Psi \vdash q : (f, \sigma)}$

$$\frac{\sigma = \tau_1 \texttt{*} \ldots \texttt{*} \tau_n \texttt{->} \tau \quad \Psi, \{x_i \mapsto \tau_i \mid i \in [1..n]\} \vdash e : \tau}{\Psi \vdash f \texttt{:} \sigma(x_1, \ldots, x_n) \texttt{=} e : (f, \sigma)}$$

$\boxed{\vdash d : \Psi}$

$$\frac{\Psi = \{f_i \mapsto \sigma_i \mid i \in [1..n]\} \quad \forall k \in [1..n].\, \Psi \vdash q_k : (f_k, \sigma_k)}{\vdash [q_1, \ldots, q_n] : \Psi}$$

$\boxed{\vdash p : \sigma}$

$$\frac{\Psi(f) = \sigma \quad \vdash d : \Psi}{\vdash \texttt{local}\, d\, \texttt{in}\, f : \sigma}$$

### 2.1.3 Semantics

Let an interpretation of the signature, $\mathcal{I}_\Sigma = (\mathcal{B}, \mathcal{L}, \mathcal{C})$, be given. We require that $\mathcal{B}$ maps each base type to a discrete cpo of values; and that $\mathcal{L}$ maps literals to values such that for all $\mathrm{l} \in \mathrm{L(b)}$, $\mathcal{L}(\mathrm{l}) \in \mathcal{B}(\mathrm{b})$. For simplicity, we assume that $\mathcal{L}$ is surjective; i.e., that for every value $v \in \mathcal{B}(\mathrm{b})$ we can find a literal $\mathrm{l} \in \mathrm{L(b)}$ such that $\mathcal{L}(\mathrm{l}) = v$.

**Meaning of types:**

$$[\![\mathrm{b}]\!] = \mathcal{B}(\mathrm{b})$$
$$[\![\texttt{bool}]\!] = \mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$$
$$[\![\tau_1 \texttt{*} \ldots \texttt{*} \tau_n \texttt{->} \tau]\!] = [[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to [\![\tau]\!]_\perp]$$

Furthermore, for all function constants $\mathrm{c} \in \mathrm{dom\,C}$, we require that $\mathcal{C}(\mathrm{c}) \in [\![\mathrm{C(c)}]\!]$. The spectrum of computational effects is here thus limited to partiality [20, 38].

**Meaning of typing environments:**

$$[\![\Gamma]\!] = \Pi_{x \in \mathrm{dom}\,\Gamma} [\![\Gamma(x)]\!] = \{\rho \mid \mathrm{dom}\,\rho = \mathrm{dom}\,\Gamma,\, \forall x \in \mathrm{dom}\,\Gamma.\, \rho(x) \in [\![\Gamma(x)]\!]\}$$
$$[\![\Psi]\!] = \Pi_{f \in \mathrm{dom}\,\Psi} [\![\Psi(f)]\!] = \{\xi \mid \mathrm{dom}\,\xi = \mathrm{dom}\,\Psi,\, \forall f \in \mathrm{dom}\,\Psi.\, \xi(f) \in [\![\Psi(f)]\!]\}$$

Note that $[\![\Psi]\!]$ is always a pointed cpo. We write $\lfloor\cdot\rfloor$ for the inclusion from $A$ to $A_\perp$; and for $f : A \to B$ with $B$ pointed, we write $\cdot \dagger f$ for $f$'s strict extension to $A_\perp$, i.e., $\perp \dagger f = \perp$ and $\lfloor a \rfloor \dagger f = f\, a$. Let further $\mathrm{fix}(\cdot)$ denote the least-fixed-point operator.

**Meaning of well-typed terms:** If $\Psi, \Gamma \vdash e : \tau$, we define $[\![e]\!] \in [[\![\Psi]\!] \to [[\![\Gamma]\!] \to [\![\tau]\!]_\perp]]$. Similarly, if $\Psi \vdash q : (f, \sigma)$ then $[\![q]\!] \in [[\![\Psi]\!] \to [\![\sigma]\!]]$; if $\vdash d : \Psi$ then $[\![d]\!] \in [\![\Psi]\!]$; and if $\vdash p : \sigma$ then $[\![p]\!] \in [\![\sigma]\!]$.

$$
\begin{aligned}
[\![\mathtt{l}]\!]\,\xi\,\rho &= \lfloor \mathcal{L}(\mathtt{l}) \rfloor \\
[\![\mathtt{true}]\!]\,\xi\,\rho &= \lfloor \mathtt{tt} \rfloor \\
[\![\mathtt{false}]\!]\,\xi\,\rho &= \lfloor \mathtt{ff} \rfloor \\
[\![\mathtt{c(}e_1\mathtt{,}\ldots\mathtt{,}e_n\mathtt{)}]\!]\,\xi\,\rho &= [\![e_1]\!]\,\xi\,\rho \dagger \lambda v_1.\ \ldots\ [\![e_n]\!]\,\xi\,\rho \dagger \lambda v_n.\, \mathcal{C}(\mathtt{c})(v_1, \ldots, v_n) \\
[\![\mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3]\!]\,\xi\,\rho &= [\![e_1]\!]\,\xi\,\rho \dagger \lambda b.\, \mathrm{case}\ b\ \mathrm{of} \begin{cases} \mathtt{tt} \to [\![e_2]\!]\,\xi\,\rho \\ \mathtt{ff} \to [\![e_3]\!]\,\xi\,\rho \end{cases} \\
[\![x]\!]\,\xi\,\rho &= \lfloor \rho(x) \rfloor \\
[\![\mathtt{let}\, x\mathtt{:}\tau_1 \Leftarrow e_1\, \mathtt{in}\, e_2]\!]\,\xi\,\rho &= [\![e_1]\!]\,\xi\,\rho \dagger \lambda v.\, [\![e_2]\!]\,\xi\,\rho[x \mapsto v] \\
[\![f\mathtt{(}x_1\mathtt{,}\ldots\mathtt{,}x_n\mathtt{)}]\!]\,\xi\,\rho &= \xi(f)(\rho(x_1), \ldots, \rho(x_n)) \\
[\![f\mathtt{:}\sigma\mathtt{(}x_1\mathtt{,}\ldots\mathtt{,}x_n\mathtt{)=}e]\!]\,\xi &= \lambda(v_1, \ldots, v_n).\, [\![e]\!]\,\xi\,\{x_i \mapsto v_i \mid i \in [1..n]\}
\end{aligned}
$$

If $\vdash \mathtt{[}q_1\mathtt{,}\ldots\mathtt{,}q_n\mathtt{]} : \Psi$ with $\forall i \in [1..n].\, \Psi \vdash q_i : (f_i, \sigma_i)$, note that $\Phi : [\![\Psi]\!] \to [\![\Psi]\!]$ below is indeed a continuous operator on a pointed cpo.

$$
\begin{aligned}
[\![\mathtt{[}q_1\mathtt{,}\ldots\mathtt{,}q_n\mathtt{]}]\!] &= \begin{pmatrix} \text{let } \Phi(\xi) = \{f_i \mapsto [\![q_i]\!]\,\xi \mid i \in [1..n]\} \\ \text{in } \mathrm{fix}(\Phi) \end{pmatrix} \\
[\![\mathtt{local}\, d\, \mathtt{in}\, f]\!] &= [\![d]\!](f)
\end{aligned}
$$

We shall need some standard weakening properties:

**Lemma 1 (weakening)** *If $\Psi, \Gamma \vdash e : \tau$ and $f \notin \mathrm{dom}\,\Psi$ then for all $\sigma \in FTyp$,*

1. $\Psi[f \mapsto \sigma], \Gamma \vdash e : \tau$
2. $\forall g \in [\![\sigma]\!].\, [\![e]\!]\,\xi\,\rho = [\![e]\!]\,\xi[f \mapsto g]\,\rho$

**Proof:** Straightforward, by structural induction. □

**Lemma 2 (weakening)** *If $\Psi \vdash q : (f', \sigma')$ and $f \notin \mathrm{dom}\,\Psi$ then for all $\sigma \in FTyp$,*

1. $\Psi[f \mapsto \sigma] \vdash q : (f', \sigma')$
2. $\forall g \in [\![\sigma]\!].\, [\![q]\!]\,\xi = [\![q]\!]\,\xi[f \mapsto g]$

**Proof:** Follows directly from Lemma 1. □

## 2.2 A binding-time-annotated language

We introduce a binding-time-annotated variant of the typed first-order language. The annotations are intended to direct the specialization algorithm. As traditional we overline static constructs (understood as "to be evaluated") and underline dynamic ones (understood as "to be residualized").

### 2.2.1 Syntax

The annotated language is given with respect to the original signature, $\Sigma = (\mathrm{B}, \mathrm{L}, \mathrm{C})$. Distinctness of argument and function names is again assumed.

**Types:**

$$
\begin{array}{lll}
\widehat{GTyp}: & \widehat{\tau} ::= \overline{\tau} \mid \underline{\tau} & \text{(Ground types)} \\
\widehat{FTyp}: & \widehat{\sigma} ::= \widehat{\tau}_1 \mathtt{*} \ldots \mathtt{*} \widehat{\tau}_n \mathtt{->} \widehat{\tau} & \text{(Function types)}
\end{array}
$$

**Terms:**

$$
\begin{array}{lll}
\widehat{Exp}: & \widehat{e} ::= & \overline{\mathrm{l}} \mid \overline{\texttt{true}} \mid \overline{\texttt{false}} \mid \overline{\mathrm{c}}(\widehat{e}_1,\ldots,\widehat{e}_n) \mid \overline{\texttt{if}}\,\widehat{e}_1\,\overline{\texttt{then}}\,\widehat{e}_2\,\overline{\texttt{else}}\,\widehat{e}_3 \mid \\
& & x \mid \overline{\texttt{let}}\,x{:}\widehat{\tau}_1 \Leftarrow \widehat{e}_1\,\overline{\texttt{in}}\,\widehat{e}_2 \mid \overline{f}(x_1,\ldots,x_n) \\
& & \$_\tau\,\widehat{e} \mid \underline{\mathrm{c}}(\widehat{e}_1,\ldots,\widehat{e}_n) \mid \underline{\texttt{if}}\,\widehat{e}_1\,\underline{\texttt{then}}\,\widehat{e}_2\,\underline{\texttt{else}}\,\widehat{e}_3 \mid \\
& & \underline{\texttt{let}}\,x{:}\widehat{\tau}_1 \Leftarrow \widehat{e}_1\,\underline{\texttt{in}}\,\widehat{e}_2 \mid \underline{f}(x_1,\ldots,x_n) \\
\widehat{Eqn}: & \widehat{q} ::= & f{:}\widehat{\sigma}(x_1,\ldots,x_n){=}\widehat{e} \\
\widehat{Decl}: & \widehat{d} ::= & [\widehat{q}_1,\ldots,\widehat{q}_n] \\
\widehat{Pgm}: & \widehat{p} ::= & \texttt{local}\,\widehat{d}\,\texttt{in}\,f
\end{array}
$$

### 2.2.2 Typing

Let $\widehat{\Gamma} : \mathit{Var} \to \widehat{GTyp}$, and $\widehat{\Psi} : \mathit{FVar} \to \widehat{FTyp}$ be typing environments. Well-typed terms (whose binding-time annotations are called *congruent* [33]) are defined analogously to before. We distinguish the three possible annotations for a function call: "to be unfolded" (evaluated), "to be *symbolically* unfolded" (reduced), or "to be residualized" (rebuilt).

$\boxed{\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e} : \widehat{\tau}}$

$$\frac{\mathrm{l} \in \mathrm{L(b)}}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{\mathrm{l}} : \overline{\mathrm{b}}}$$

$$\frac{}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{\texttt{true}} : \overline{\texttt{bool}}}$$

$$\frac{}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{\texttt{false}} : \overline{\texttt{bool}}}$$

$$\frac{\mathrm{C(c)} = \tau_1 * \ldots * \tau_n \texttt{->} \tau \quad \forall i \in [1..n].\, \widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_i : \overline{\tau}_i}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{\mathrm{c}}(\widehat{e}_1,\ldots,\widehat{e}_n) : \overline{\tau}}$$

$$\frac{\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_1 : \overline{\texttt{bool}} \quad \widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_2 : \widehat{\tau} \quad \widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_3 : \widehat{\tau}}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{\texttt{if}}\,\widehat{e}_1\,\overline{\texttt{then}}\,\widehat{e}_2\,\overline{\texttt{else}}\,\widehat{e}_3 : \widehat{\tau}}$$

$$\frac{\widehat{\Gamma}(x) = \widehat{\tau}}{\widehat{\Psi},\widehat{\Gamma} \vdash x : \widehat{\tau}}$$

$$\frac{\widehat{\Psi}(f) = \overline{\tau}_1 * \ldots * \overline{\tau}_n \texttt{->} \overline{\tau} \quad \forall i \in [1..n].\, \widehat{\Gamma}(x_i) = \overline{\tau}_i}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{f}(x_1,\ldots,x_n) : \overline{\tau}}$$

$$\frac{\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_1 : \overline{\tau}_1 \quad \widehat{\Psi},\widehat{\Gamma}[x \mapsto \overline{\tau}_1] \vdash \widehat{e}_2 : \widehat{\tau}_2}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{\texttt{let}}\,x{:}\overline{\tau}_1 \Leftarrow \widehat{e}_1\,\overline{\texttt{in}}\,\widehat{e}_2 : \widehat{\tau}_2}$$

$$\frac{\widehat{\Psi}(f) = \widehat{\tau}_1 * \ldots * \widehat{\tau}_n \texttt{->} \underline{\tau} \quad \forall i \in [1..n].\, \widehat{\Gamma}(x_i) = \widehat{\tau}_i}{\widehat{\Psi},\widehat{\Gamma} \vdash \overline{f}(x_1,\ldots,x_n) : \underline{\tau}}$$

$$\frac{\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e} : \overline{\tau}}{\widehat{\Psi},\widehat{\Gamma} \vdash \$_\tau\,\widehat{e} : \underline{\tau}}$$

$$\frac{\mathrm{C(c)} = \tau_1 * \ldots * \tau_n \texttt{->} \tau \quad \forall i \in [1..n].\, \widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_i : \underline{\tau}_i}{\widehat{\Psi},\widehat{\Gamma} \vdash \underline{\mathrm{c}}(\widehat{e}_1,\ldots,\widehat{e}_n) : \underline{\tau}}$$

$$\frac{\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_1 : \underline{\texttt{bool}} \quad \widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_2 : \underline{\tau} \quad \widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_3 : \underline{\tau}}{\widehat{\Psi},\widehat{\Gamma} \vdash \underline{\texttt{if}}\,\widehat{e}_1\,\underline{\texttt{then}}\,\widehat{e}_2\,\underline{\texttt{else}}\,\widehat{e}_3 : \underline{\tau}}$$

$$\frac{\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e}_1 : \underline{\tau}_1 \quad \widehat{\Psi},\widehat{\Gamma}[x \mapsto \underline{\tau}_1] \vdash \widehat{e}_2 : \underline{\tau}_2}{\widehat{\Psi},\widehat{\Gamma} \vdash \underline{\texttt{let}}\,x{:}\underline{\tau}_1 \Leftarrow \widehat{e}_1\,\underline{\texttt{in}}\,\widehat{e}_2 : \underline{\tau}_2}$$

$$\frac{\widehat{\Psi}(f) = \widehat{\tau}_1 * \ldots * \widehat{\tau}_n \texttt{->} \underline{\tau} \quad \forall i \in [1..n].\, \widehat{\Gamma}(x_i) = \widehat{\tau}_i}{\widehat{\Psi},\widehat{\Gamma} \vdash \underline{f}(x_1,\ldots,x_n) : \underline{\tau}}$$

$\boxed{\widehat{\Psi} \vdash \widehat{q} : (f, \widehat{\sigma})}$

$$\frac{\widehat{\sigma} = \widehat{\tau}_1 \texttt{*} \ldots \texttt{*} \widehat{\tau}_n \texttt{->} \widehat{\tau} \quad \widehat{\Psi}, \{x_i \mapsto \widehat{\tau}_i \mid i \in [1..n]\} \vdash \widehat{e} : \widehat{\tau}}{\widehat{\Psi} \vdash f \texttt{:} \widehat{\sigma} \texttt{(} x_1 \texttt{,} \ldots \texttt{,} x_n \texttt{)=} \widehat{e} : (f, \widehat{\sigma})}$$

$\boxed{\vdash \widehat{d} : \widehat{\Psi}}$

$$\frac{\widehat{\Psi} = \{f_i \mapsto \widehat{\sigma}_i \mid i \in [1..n]\} \quad \forall k \in [1..n].\, \widehat{\Psi} \vdash \widehat{q}_k : (f_k, \widehat{\sigma}_k)}{\vdash \texttt{[} \widehat{q}_1 \texttt{,} \ldots \texttt{,} \widehat{q}_n \texttt{]} : \widehat{\Psi}}$$

$\boxed{\vdash \widehat{p} : \widehat{\sigma}}$

$$\frac{\widehat{\Psi}(f) = \widehat{\sigma} = \widehat{\tau}_1 \texttt{*} \ldots \texttt{*} \widehat{\tau}_n \texttt{->} \underline{\tau} \quad \vdash \widehat{d} : \widehat{\Psi}}{\vdash \texttt{local}\, \widehat{d}\, \texttt{in}\, f : \widehat{\sigma}}$$

Note that the typing rules disallow functions to be annotated with a static return type but dynamic arguments, since the dynamic arguments cannot currently be used in any meaningful way. In the presence of bounded static variation [33], however, this situation would change (see also Section 5.2).

### 2.2.3 Semantics

The intended standard semantics of the binding-time-separated language is virtually identical to the previous semantics, simply ignoring binding-time annotations and interpreting \$ as identity.

This can be straightforwardly formalized as a family of erasure translations, $\|\cdot\|$, on types, terms, and typing environments, taking each construct to its unannotated counterpart. For example,

$$\|\underline{\texttt{let}}\, x \texttt{:} \underline{\tau}_1 \Leftarrow \widehat{e}_1\, \underline{\texttt{in}}\, \widehat{e}_2\| = \texttt{let}\, x \texttt{:} \tau_1 \Leftarrow \|\widehat{e}_1\|\, \texttt{in}\, \|\widehat{e}_2\| \quad \text{and} \quad \|\$_\tau\, \widehat{e}\| = \|\widehat{e}\|.$$

It is easy to show that well-typedness is preserved. The meaning of each annotated construct is then simply the meaning of its erasure.

**The role of binding-time analysis** In our formulation, a (monovariant) binding-time analyzer, $\text{bta}(\cdot) : Pgm \times \widehat{FTyp} \to \widehat{Pgm}$, is thus a function satisfying that if $\widehat{\sigma} = \widehat{\tau}_1 \texttt{*} \ldots \texttt{*} \widehat{\tau}_n \texttt{->} \underline{\tau}$ and $\vdash p : \|\widehat{\sigma}\|$ then $\vdash \text{bta}(p, \widehat{\sigma}) : \widehat{\sigma}$ and $\|\text{bta}(p, \widehat{\sigma})\| = p$. In particular, a binding-time analyzer must preserve meanings.

Of course, one can also write programs in $\widehat{Pgm}$ directly; writing such programs is for example done when writing macros [5].

## 2.3 The specialization algorithm

We define polyvariant specialization as a non-standard, residualizing semantics.

### 2.3.1 Notation

For residualized function calls, we need some notation that lets us separate static and dynamic arguments. Given $\widehat{\sigma} = \widehat{\tau}_1 \texttt{*} \ldots \texttt{*} \widehat{\tau}_n \texttt{->} \underline{\tau}$, we define the *index* of the $i$'th static argument, $(i)_{\overline{\widehat{\sigma}}}$, and of the $i$'th dynamic argument, $(i)_{\underline{\widehat{\sigma}}}$. For example, if $\widehat{\sigma} = \overline{\text{b}}_1 \texttt{*} \underline{\text{b}}_2 \texttt{*} \overline{\text{b}}_3 \texttt{->} \underline{\text{b}}$ then $(1)_{\overline{\widehat{\sigma}}} = 1$,$(2)_{\overline{\widehat{\sigma}}} = 3$, and $(1)_{\underline{\widehat{\sigma}}} = 2$. We formalize these indices as follows. Let first the predicates

$$\begin{array}{ll} S(\widehat{\tau}) \Leftrightarrow (\exists \tau.\, \widehat{\tau} = \overline{\tau}), & S(\widehat{\sigma}) \Leftrightarrow (\exists \widehat{\tau}_1, \ldots, \widehat{\tau}_n, \tau.\, \widehat{\sigma} = \widehat{\tau}_1 \texttt{*} \ldots \texttt{*} \widehat{\tau}_n \texttt{->} \overline{\tau}) \\ D(\widehat{\tau}) \Leftrightarrow (\exists \tau.\, \widehat{\tau} = \underline{\tau}), & D(\widehat{\sigma}) \Leftrightarrow (\exists \widehat{\tau}_1, \ldots, \widehat{\tau}_n, \tau.\, \widehat{\sigma} = \widehat{\tau}_1 \texttt{*} \ldots \texttt{*} \widehat{\tau}_n \texttt{->} \underline{\tau}) \end{array}$$

assert that a given type is static or dynamic, respectively. For any function type $\widehat{\sigma} = \widehat{\tau}_1 \texttt{*} \ldots \texttt{*} \widehat{\tau}_n \texttt{->} \underline{\tau}$, let $|\overline{\widehat{\sigma}}| = |\{i \in [1..n] \mid S(\widehat{\tau}_i)\}|$ and $|\underline{\widehat{\sigma}}| = |\{i \in [1..n] \mid D(\widehat{\tau}_i)\}|$ be the number of static and dynamic arguments, resp. Note that it is always the case that $|\overline{\widehat{\sigma}}| + |\underline{\widehat{\sigma}}| = n$. Define then

$$\begin{array}{ll} \forall i \in [1..|\overline{\widehat{\sigma}}|],\ (i)_{\overline{\widehat{\sigma}}} & = \min\, k \in [1..n] \text{ s.t. } (|\{j \in [1..k] \mid S(\widehat{\tau}_j)\}| = i) \\ \forall i \in [1..|\underline{\widehat{\sigma}}|],\ (i)_{\underline{\widehat{\sigma}}} & = \min\, k \in [1..n] \text{ s.t. } (|\{j \in [1..k] \mid D(\widehat{\tau}_j)\}| = i) \end{array}$$

Given a function type $\widehat{\sigma}$, let further $\mathrm{curry}_{\widehat{\sigma}}$ generalize the standard binary 'curry' combinator, i.e., $\mathrm{curry} = \lambda f.\lambda x.\lambda y.f(x,y)$, to static and dynamic arguments. For example, if $\widehat{\sigma} = \overline{\mathrm{b}_1}*\underline{\mathrm{b}_2}*\overline{\mathrm{b}_3}\texttt{->}\underline{\mathrm{b}}$ then $\mathrm{curry}_{\widehat{\sigma}} = \lambda f.\,\lambda(x_1,x_3).\,\lambda x_2.\,f(x_1,x_2,x_3)$. We define $\mathrm{curry}_{\widehat{\sigma}}$ by

$$\begin{array}{c}\mathrm{curry}_{\widehat{\sigma}} = \lambda f.\,\lambda(v_1,\ldots,v_k).\,\lambda(w_1,\ldots,w_{k'}).\,f(u_1,\ldots,u_n)\\ \text{where } \forall j\in[1..n].\,(u_j = v_i \Leftrightarrow (i)_{\overline{\widehat{\sigma}}} = j) \wedge (u_j = w_i \Leftrightarrow (i)_{\underline{\widehat{\sigma}}} = j)\end{array}$$

Now, given a function type $\widehat{\sigma} = \widehat{\tau}_1*\ldots*\widehat{\tau}_n\texttt{->}\underline{\tau}$, let

$$\underline{\widehat{\sigma}} = \widehat{\tau}_{(1)_{\underline{\widehat{\sigma}}}}*\ldots*\widehat{\tau}_{(|\underline{\widehat{\sigma}}|)_{\underline{\widehat{\sigma}}}}\texttt{->}\underline{\tau}$$

be its residual type (i.e., $\widehat{\sigma}$ without the static arguments). Also, let (by slight abuse of notation, since we do not have product types)

$$[\![\|\overline{\widehat{\sigma}}\|]\!] = [\![\|\widehat{\tau}_{(1)_{\overline{\widehat{\sigma}}}}\|]\!] \times \ldots \times [\![\|\widehat{\tau}_{(|\overline{\widehat{\sigma}}|)_{\overline{\widehat{\sigma}}}}\|]\!]$$

be the set of $\widehat{\sigma}$'s evaluated static arguments.

We assume convenient names for specialized functions, i.e., $f_{(v_1,\ldots,v_k)}$, in $FVar$. Let

$$FVar_{\widehat{\Psi}} = \{f_{(v_1,\ldots,v_k)} \mid f \in \mathrm{dom}\,\widehat{\Psi}, (v_1,\ldots,v_k) \in [\![\|\overline{\widehat{\Psi}(f)}\|]\!]\}$$

be the well-formed residual function names in the context of $\widehat{\Psi}$.

In the polyvariant specializer, the residualized function calls will refer to such residual function names. Hence, we need some notation to specify the contexts in which residualized terms are to be given meaning. We extend the notions of residual function types and currying to environments. If $\widehat{\Psi}$, $X \subseteq_{\mathrm{fin}} FVar_{\widehat{\Psi}}$, and $\xi \in [\![\|\widehat{\Psi}\|]\!]$, let

$$\begin{array}{rl}\underline{\widehat{\Psi}}(X) &= \{f_{(v_1,\ldots,v_k)} \mapsto \underline{\widehat{\Psi}(f)} \mid D(\widehat{\Psi}(f)), f_{(v_1,\ldots,v_k)} \in X\}\\ \mathrm{curry}_{\widehat{\Psi}}(\xi)(X) &= \{f_{(v_1,\ldots,v_k)} \mapsto \mathrm{curry}_{\widehat{\Psi}(f)}(\xi(f))(v_1,\ldots,v_k) \mid D(\widehat{\Psi}(f)), f_{(v_1,\ldots,v_k)} \in X\}\end{array}$$

Note that $\mathrm{curry}_{\widehat{\Psi}}(\xi)(X) \in [\![\|\underline{\widehat{\Psi}}(X)\|]\!]$.

### 2.3.2 State and effects

The residualizing semantics is phrased in so-called monadic style. We use a specific state monad to support the bookkeeping involved, layered on top of partiality (recall that partiality is the computational effect of the original language). A notable simplification of this bookkeeping is the convenient names for specialized functions, i.e., $f_{(v_1,\ldots,v_k)}$, in $FVar$. Hence, we can recover the static argument values from the name of a specialized function. Accordingly, let

$$\Delta = \{\{\texttt{i} \mapsto i, \texttt{seen} \mapsto s, \texttt{pen} \mapsto p, \texttt{res} \mapsto d\} \mid i \in \mathbb{N}, s \in \mathcal{P}_{fin}(FVar), p \in FVar\,\mathrm{list}, d \in Decl\}$$

be the cpo of states, where $\mathcal{P}_{fin}(FVar)$ is discretely ordered. Hence, $\Delta$ is a discrete cpo. The intended reading is that `i` is a counter for generating "fresh" names; `seen` is the set of (names of) visited equations (the so-called 'déjà-vu' or 'seen-before' list); `pen` is the worklist of (names of) visited but not generated equations (the so-called 'pending' list); and `res` is the generated equations.

We will find it convenient to allow manipulation of declarations as if they were lists. Also, if $d = \texttt{[}q_1\texttt{,}\ldots\texttt{,}q_n\texttt{]}$, and $q_i = f_i\texttt{:}\sigma_i\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{)=}e_i$ for all $i\in[1..n]$, let $\mathrm{dom}\,d = \{f_1,\ldots,f_n\}$ denote the set of function names.

For all cpos $A$ and $B$ and $f \in [A \to TB]$, we define

$$\begin{array}{rl}TA & = [\Delta \to (A \times \Delta)_\perp]\\ \eta_A : A \to TA & = \lambda a.\,\lambda\delta.\,\lfloor(a,\delta)\rfloor\\ \cdot\star f : TA \to TB & = \lambda t.\,\lambda\delta.\,(t\,\delta \dagger \lambda(a,\delta').\,f\,a\,\delta')\end{array}$$

For static computations we need to transfer answers. This is accomplished via the monad morphism $\gamma_T : A_\perp \to TA$ defined as $\gamma_T(v) = v \dagger \eta$. Note that if $t\ \delta = \lfloor (a, \delta) \rfloor$ and $t = \gamma_T(v)$ then $v = \lfloor a \rfloor$. For the purpose of unique name generation, let $\{y_0, y_1, ...\}$ be a countably-infinite subset of $Var$, such that $y_i = y_j$ implies $i = j$.

We define

$$\begin{aligned}
\text{new} : T\,Var &= \lambda\delta.\, \lfloor (y_{\delta(\mathtt{i})}, \delta[\mathtt{i} \mapsto \delta(\mathtt{i}) + 1]) \rfloor \\
\text{visit} : FVar \to T\mathbf{1} &= \lambda f.\, \lambda\delta.\, \text{case } f \in \delta(\mathtt{seen}) \text{ of } \begin{cases} \mathtt{tt} \to \lfloor (*, \delta) \rfloor \\ \mathtt{ff} \to \lfloor (*, \delta \left[ \begin{array}{l} \mathtt{seen} \mapsto \delta(\mathtt{seen}) \cup \{f\}, \\ \mathtt{pen} \mapsto f{::}\delta(\mathtt{pen}) \end{array} \right]) \rfloor \end{cases} \\
\text{commit} : Eqn \to T\mathbf{1} &= \lambda q.\, \lambda\delta.\, \lfloor (*, \delta[\mathtt{res} \mapsto q{::}\delta(\mathtt{res})]) \rfloor \\
\text{next} : T(\mathbf{1} + FVar) &= \lambda\delta.\, \text{case } \delta(\mathtt{pen}) \text{ of } \begin{cases} f{::}X \to \lfloor (\text{in}_2(f), \delta[\mathtt{pen} \mapsto X]) \rfloor \\ nil \to \lfloor (\text{in}_1(*), \delta) \rfloor \end{cases} \\
\text{result} : T\,Decl &= \lambda\delta.\, \lfloor (\delta(\mathtt{res}), \delta) \rfloor
\end{aligned}$$

These operations will provide adequate support for the bookkeeping involved with the polyvariant element of specialization. new returns a "fresh" variable name; visit($f$) schedules $f$ for residualization (if it has not been already); commit($q$) appends $q$ to the result (where the name of $q$ is assumed to have been visited); next returns (and removes) the head of the pending list; and result returns the generated (i.e., committed) equations.

### 2.3.3 The residualizing semantics

Recall that in the source language, arguments to function calls are required to be syntactic variables. We slightly restrict the meaning of types to ensure this. Define first:

$$\begin{aligned}
[\![\overline{\tau}]\!]_{\mathsf{V}} &= [\![\tau]\!] \\
[\![\underline{\tau}]\!]_{\mathsf{V}} &= Var
\end{aligned}$$

**Meaning of types:**

$$\begin{aligned}
[\![\overline{\tau}]\!]_{\mathsf{R}} &= [\![\tau]\!] \\
[\![\underline{\tau}]\!]_{\mathsf{R}} &= Exp \\
[\![\widehat{\tau}_1 \mathtt{*} \ldots \mathtt{*} \widehat{\tau}_n \mathtt{->} \widehat{\tau}]\!]_{\mathsf{R}} &= [[\![\widehat{\tau}_1]\!]_{\mathsf{V}} \times \ldots \times [\![\widehat{\tau}_n]\!]_{\mathsf{V}} \to T[\![\widehat{\tau}]\!]_{\mathsf{R}}]
\end{aligned}$$

**Meaning of typing environments:**

$$\begin{aligned}
[\![\widehat{\Gamma}]\!]_{\mathsf{R}} = \Pi_{x \in \mathrm{dom}\,\widehat{\Gamma}} [\![\widehat{\Gamma}(x)]\!]_{\mathsf{R}} &= \{\widehat{\rho} \mid \mathrm{dom}\,\widehat{\rho} = \mathrm{dom}\,\widehat{\Gamma}, \forall x \in \mathrm{dom}\,\widehat{\Gamma}.\, \widehat{\rho}(x) \in [\![\widehat{\Gamma}(x)]\!]_{\mathsf{V}}\} \\
[\![\widehat{\Psi}]\!]_{\mathsf{R}} = \Pi_{f \in \mathrm{dom}\,\widehat{\Psi}} [\![\widehat{\Psi}(f)]\!]_{\mathsf{R}} &= \{\widehat{\xi} \mid \mathrm{dom}\,\widehat{\xi} = \mathrm{dom}\,\widehat{\Psi}, \forall f \in \mathrm{dom}\,\widehat{\Psi}.\, \widehat{\xi}(f) \in [\![\widehat{\Psi}(f)]\!]_{\mathsf{R}}\}
\end{aligned}$$

We previously assumed that each base value was represented by a literal (an extension to booleans is trivial); we now let $\downarrow_\tau (\cdot) : [\![\tau]\!] \to Exp$ pick particular such literals, i.e., for every $a \in [\![\tau]\!]$, if $\downarrow_\tau (a) = \mathrm{l}$ then $[\![\mathrm{l}]\!]\,\xi\,\rho = \lfloor a \rfloor$.

**Meaning of well-typed terms:** If $\widehat{\Psi}, \widehat{\Gamma} \vdash \widehat{e} : \widehat{\tau}$, we define $[\![\widehat{e}]\!]_{\mathsf{R}} \in [[\![\widehat{\Psi}]\!]_{\mathsf{R}} \to [[\![\widehat{\Gamma}]\!]_{\mathsf{R}} \to T[\![\widehat{\tau}]\!]_{\mathsf{R}}]]$. Note that the only effectful operations used are new and visit.

$$\begin{aligned}
[\![\overline{\mathrm{l}}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= \eta(\mathcal{L}(\mathrm{l})) \\
[\![\overline{\mathtt{true}}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= \eta(\mathtt{tt}) \\
[\![\overline{\mathtt{false}}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= \eta(\mathtt{ff}) \\
[\![\overline{\mathtt{c}}(\widehat{e}_1, \ldots, \widehat{e}_n)]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= [\![\widehat{e}_1]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda v_1.\ \ldots\ [\![\widehat{e}_n]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda v_n. \\
&\qquad \gamma_T(\mathcal{C}(\mathrm{c})(v_1, \ldots, v_n)) \\
[\![\overline{\mathtt{if}}\,\widehat{e}_1\,\overline{\mathtt{then}}\,\widehat{e}_2\,\overline{\mathtt{else}}\,\widehat{e}_3]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= [\![\widehat{e}_1]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda b.\, \text{case } b \text{ of } \begin{cases} \mathtt{tt} \to [\![\widehat{e}_2]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \\ \mathtt{ff} \to [\![\widehat{e}_3]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \end{cases} \\
[\![x]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= \eta(\widehat{\rho}(x)) \\
[\![\overline{\mathtt{let}}\,x{:}\widehat{\tau}_1 \Leftarrow \widehat{e}_1\,\overline{\mathtt{in}}\,\widehat{e}_2]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= [\![\widehat{e}_1]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda v.\, [\![\widehat{e}_2]\!]_{\mathsf{R}}\,\widehat{\rho}[x \mapsto v]
\end{aligned}$$

$$[\![\overline{f}(x_1,\ldots,x_n)]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} = \widehat{\xi}(f)(\widehat{\rho}(x_1),\ldots,\widehat{\rho}(x_n))$$

$$\begin{aligned}
[\![\$_\tau\,\widehat{e}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= [\![\widehat{e}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda v.\,\eta(\downarrow_\tau(v))\\
[\![\underline{\mathtt{c}}(\widehat{e}_1,\ldots,\widehat{e}_n)]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= [\![\widehat{e}_1]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda e_1.\ \ldots\ [\![\widehat{e}_n]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda e_n.\\
&\qquad \eta(\mathtt{c}(e_1,\ldots,e_n))\\
[\![\underline{\mathtt{if}}\,\widehat{e}_1\,\underline{\mathtt{then}}\,\widehat{e}_2\,\underline{\mathtt{else}}\,\widehat{e}_3]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= [\![\widehat{e}_1]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda e_1.\,[\![\widehat{e}_2]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda e_2.\,[\![\widehat{e}_3]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda e_3.\\
&\qquad \eta(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3)\\
[\![\underline{\mathtt{let}}\,x{:}\underline{\tau}_1 \Leftarrow \widehat{e}_1\,\underline{\mathtt{in}}\,\widehat{e}_2]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} &= \mathrm{new} \star \lambda y.\,[\![\widehat{e}_1]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} \star \lambda e_1.\,[\![\widehat{e}_2]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho}[x \mapsto y] \star \lambda e_2.\\
&\qquad \eta(\mathtt{let}\ y{:}\tau_1 \Leftarrow e_1\ \mathtt{in}\ e_2)
\end{aligned}$$

If $\widehat{\Psi},\widehat{\Gamma} \vdash \underline{f}(x_1,\ldots,x_n) : \underline{\tau}$ with $\widehat{\Psi}(f) = \widehat{\sigma}$ then

$$\begin{aligned}
[\![\underline{f}(x_1,\ldots,x_n)]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} = \mathrm{visit}(f_{(\widehat{\rho}(x_{(1)_{\overline{\widehat{\sigma}}}}),\ldots,\widehat{\rho}(x_{(|\overline{\widehat{\sigma}}|)_{\overline{\widehat{\sigma}}}}))}) \star \lambda u.\\
\eta(f_{(\widehat{\rho}(x_{(1)_{\overline{\widehat{\sigma}}}}),\ldots,\widehat{\rho}(x_{(|\overline{\widehat{\sigma}}|)_{\overline{\widehat{\sigma}}}}))}(\widehat{\rho}(x_{(1)_{\underline{\widehat{\sigma}}}}),\ldots,\widehat{\rho}(x_{(|\underline{\widehat{\sigma}}|)_{\underline{\widehat{\sigma}}}})))
\end{aligned}$$

Recall above that $\widehat{\rho}(x_{(1)_{\overline{\widehat{\sigma}}}})\ldots\widehat{\rho}(x_{(|\overline{\widehat{\sigma}}|)_{\overline{\widehat{\sigma}}}})$ is simply the values of the static arguments, and $\widehat{\rho}(x_{(1)_{\underline{\widehat{\sigma}}}})$ $\ldots$ $\widehat{\rho}(x_{(|\underline{\widehat{\sigma}}|)_{\underline{\widehat{\sigma}}}})$ are the values (which must be variables due to $[\![\cdot]\!]_{\mathsf{V}}$) of the dynamic arguments.

If $\widehat{\Psi} \vdash \widehat{q} : (f,\widehat{\sigma})$ we define $[\![\widehat{q}]\!]_{\mathsf{R}} \in [[\![\widehat{\Psi}]\!]_{\mathsf{R}} \to [\![\widehat{\sigma}]\!]_{\mathsf{R}}]$ by

$$[\![f{:}\widehat{\sigma}(x_1,\ldots,x_n){=}\widehat{e}]\!]_{\mathsf{R}}\,\widehat{\xi} = \lambda(v_1,\ldots,v_n).\,[\![\widehat{e}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\{x_i \mapsto v_i \mid i \in [1..n]\}$$

Since the residualized meaning of a dynamic equation is an expression-generating function, we shall need a way to convert it to an equation. All we need is a function name, some distinct variables (fresh variables are fine), and some values: if $D(\widehat{\sigma})$ and $g \in [\![\widehat{\sigma}]\!]_{\mathsf{R}}$, we define $\downarrow_{\widehat{\sigma}}(g)(\cdot,\cdot) \in [(FVar \times [\![\|\overline{\widehat{\sigma}}\|]\!]) \to TEqn]$ by

$$\begin{aligned}
\downarrow_{\widehat{\sigma}}(g)(f,(v_1,\ldots,v_k)) = \mathrm{new} \star \lambda x_1.\ \ldots \mathrm{new} \star \lambda x_{k'}.\,\mathrm{curry}_{\widehat{\sigma}}(g)(v_1,\ldots,v_k)(x_1,\ldots,x_{k'}) \star \lambda e.\\
\eta(f{:}\underline{\sigma}(x_1,\ldots,x_{k'}){=}e)
\end{aligned}$$

The output, if any, is the corresponding residual equation (note that the use of $\mathrm{curry}_{\widehat{\sigma}}(g)$ is inessential, it is only a convenient way to properly merge the static and dynamic arguments).

If $\vdash [\![\widehat{q}_1,\ldots,\widehat{q}_n]\!] : \widehat{\Psi}$ where $(\forall i \in [1..n].\,\widehat{\Psi} \vdash \widehat{q}_i : (f_i,\widehat{\sigma}_i))$, we define $[\![\widehat{d}]\!]_{\mathsf{R}} \in [\![\widehat{\Psi}]\!]_{\mathsf{R}}$ by

$$[\![[\![\widehat{q}_1,\ldots,\widehat{q}_n]\!]]\!]_{\mathsf{R}} = \begin{pmatrix}\text{let } \tilde{\Phi}(\widehat{\xi}) = \{f_i \mapsto [\![\widehat{q}_i]\!]_{\mathsf{R}}\,\widehat{\xi} \mid i \in [1..n]\}\\ \text{in } \ \mathrm{fix}(\tilde{\Phi})\end{pmatrix}$$

We can now define our polyvariant specialization algorithm. The central part is the computation of the transitive closure of visited functions.

If $\widehat{\xi} \in [\![\widehat{\Psi}]\!]_{\mathsf{R}}$, the transitive closure, $\mathrm{close}_{\widehat{\Psi}}(\widehat{\xi}) : TDecl$, is defined as follows. Let $\Phi$ be the underlying operator:

$$\Phi(F) = \mathrm{next} \star \lambda s.\,\mathrm{case}\ s\ \mathrm{of}\ \begin{cases}\mathrm{in}_1(*) & \to \mathrm{result}\\ \mathrm{in}_2(f_{(v_1,\ldots,v_k)}) & \to \downarrow_{\widehat{\Psi}(f)}(\widehat{\xi}(f))(f_{(v_1,\ldots,v_k)},(v_1,\ldots,v_k)) \star \lambda q.\\ & \qquad \mathrm{commit}(q) \star F\end{cases}$$

Define then

$$\mathrm{close}_{\widehat{\Psi}}(\widehat{\xi}) = \mathrm{fix}(\Phi)$$

It is a simple worklist algorithm, where the pending list is processed. The output, if any, is a declaration consisting of a self-contained finite set of specialized functions.

Let $\delta_0 = \{\mathtt{i} \mapsto 0, \mathtt{seen} \mapsto \emptyset, \mathtt{pen} \mapsto \mathrm{nil}, \mathtt{res} \mapsto \mathrm{nil}\}$ be the initial state.
If $\vdash \widehat{p} : \widehat{\sigma}$, we define $[\![\widehat{p}]\!]_{\mathsf{R}} \in [[\![\|\widehat{\sigma}\|]\!] \to Pgm_\perp]$ by

$$\begin{aligned}
[\![\mathtt{local}\,\widehat{d}\,\mathtt{in}\,f]\!]_{\mathsf{R}}\,(v_1,\ldots,v_k) =\\
(\mathrm{visit}(f_{(v_1,\ldots,v_k)}) \star \lambda u.\,\mathrm{close}_{\widehat{\Psi}}([\![\widehat{d}]\!]_{\mathsf{R}}))\,\delta_0 \dagger \lambda(d,\delta).\,\lfloor \mathtt{local}\,d\,\mathtt{in}\,f_{(v_1,\ldots,v_k)} \rfloor
\end{aligned}$$

We will show our putative polyvariant specializer to be sound; i.e., if $\vdash \widehat{p} : \widehat{\sigma}$ and $[\![\widehat{p}]\!]_{\mathsf{R}}(v_1,\ldots,v_k) = \lfloor p \rfloor$ then $\vdash p : \widehat{\underline{\sigma}}$ and $[\![p]\!] = \mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{p}\|]\!])(v_1,\ldots,v_k)$. Since the specializer blindly follows annotations, it may easily diverge. Under certain conditions, however, we can guarantee termination.

## 2.4 Correctness of the algorithm

The main contribution of this work, namely the proof, is quite technical. This is large due to the two major complications: keeping track of the transitive closure and taming symbolic unfolding.

### 2.4.1 The transitive closure

We need to keep tight control with the state, as it is passed around, to prove that $\text{close}_{\widehat{\Psi}}(\widehat{\xi})$ will actually compute a transitive closure. Informally, everything will run smoothly if (1) the generation of the equations uses only new and visit (i.e., does not corrupt the bookkeeping); and (2) the initial state is not already corrupted.

To specify that only new and visit are used, we define the set of *simple extensions* of a state,

$$\text{ext}_{\widehat{\Psi}}(\delta) = \left\{ \delta' \;\middle|\; \begin{array}{l} \exists i \geq \delta(\mathtt{i}).\, \exists f_1, \ldots, f_n \in FVar_{\widehat{\Psi}}, n \geq 0. \\ (\text{visit}(f_1) \star \lambda u.\ \ldots \star \text{visit}(f_n))\ \delta[\mathtt{i} \mapsto i] = \lfloor (*, \delta') \rfloor \end{array} \right\}$$

Note that simple extensions are transitive: if $\delta' \in \text{ext}_{\widehat{\Psi}}(\delta)$ and $\delta'' \in \text{ext}_{\widehat{\Psi}}(\delta')$ then $\delta'' \in \text{ext}_{\widehat{\Psi}}(\delta)$.

We define a state $\delta$ to be well-formed, written $\text{well}_{\widehat{\Psi}}(\delta)$, if the elements of $\delta(\mathtt{pen})$, $P$, are all distinct, $P \cup \text{dom}\,\delta(\mathtt{res}) = \delta(\mathtt{seen}) \subseteq FVar_{\widehat{\Psi}}$, and $P \cap \text{dom}\,\delta(\mathtt{res}) = \emptyset$. For any typing context $\widehat{\Psi}$, the initial state is well-formed; and so are its simple extensions: if $\delta \in \text{ext}_{\widehat{\Psi}}(\delta_0)$ then $\text{well}_{\widehat{\Psi}}(\delta)$.

Operationally, the computation of $\text{close}_{\widehat{\Psi}}(\widehat{\xi})$ can be thought of as progressing through a series of well-formed states, which we will call the *well-formed extensions*, defined by

$$\text{wext}_{\widehat{\Psi}}(\delta) = \left\{ \delta' \;\middle|\; \begin{array}{l} \exists \delta'' \in \text{ext}_{\widehat{\Psi}}(\delta), q_1, \ldots, q_n, n \geq 0, \text{well}_{\widehat{\Psi}}(\delta), \text{well}_{\widehat{\Psi}}(\delta'), \text{well}_{\widehat{\Psi}}(\delta''). \ \#\,\delta''(\mathtt{pen}) \geq n \\ \quad \wedge\ (\text{next} \star \lambda s_1.\, \text{commit}(q_1) \ldots \text{next} \star \lambda s_n.\, \text{commit}(q_n))\ \delta'' = \lfloor (*, \delta') \rfloor \end{array} \right\}$$

where $\#\,l$ denotes the length of a list $l$. Also these extensions are transitive: if $\delta' \in \text{wext}_{\widehat{\Psi}}(\delta)$ and $\delta'' \in \text{wext}_{\widehat{\Psi}}(\delta')$ then $\delta'' \in \text{wext}_{\widehat{\Psi}}(\delta)$.

We will now prove that if $\text{close}_{\widehat{\Psi}}(\widehat{\xi})$ terminates then the result is indeed a transitive closure.

**Lemma 3** *If* $\vdash \widehat{d} : \widehat{\Psi}$, $(\text{visit}(f) \star \lambda u.\, \text{close}_{\widehat{\Psi}}(\widehat{\xi}))\ \delta_0 = \lfloor (d, \delta') \rfloor$, *and*

$$\forall q, f, (v_1, \ldots, v_k), \delta'', \delta'''.\ (\downarrow_{\widehat{\Psi}(f)} (\widehat{\xi}(f))(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor) \Rightarrow \delta''' \in \text{ext}_{\widehat{\Psi}}(\delta'')$$

*then* $\delta' \in \text{wext}_{\widehat{\Psi}}(\delta_0)$, $\delta'(\mathtt{res}) = d$, $\delta'(\mathtt{seen}) = \text{dom}\, d$, *and* $\forall q \in \delta'(\mathtt{res}).\, \exists f, (v_1, \ldots, v_k), \delta'', \delta'''.$

1. $\downarrow_{\widehat{\Psi}(f)} (\widehat{\xi}(f))(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor$
2. $\delta' \in \text{wext}_{\widehat{\Psi}}(\delta''')$.

**Proof:** By fixed-point induction. Define the predicate $P \subseteq TDecl$ by

$$P = \left\{ F \;\middle|\; \begin{array}{l} \forall \widehat{d}, \widehat{\Psi}, d, \delta, \delta', \widehat{\xi}. \\ \quad \text{if } \vdash \widehat{d} : \widehat{\Psi}, \text{well}_{\widehat{\Psi}}(\delta), F(\delta) = \lfloor (d, \delta') \rfloor, \text{ and} \\ \quad\quad a.\ \forall q, f, (v_1, \ldots, v_k), \delta'', \delta'''. \\ \quad\quad\quad (\downarrow_{\widehat{\Psi}(f)} (\widehat{\xi}(f))(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor) \Rightarrow \delta''' \in \text{ext}_{\widehat{\Psi}}(\delta'') \\ \quad\quad b.\ \forall q \in \delta(\mathtt{res}).\, \exists f, (v_1, \ldots, v_k), \delta'', \delta'''. \\ \quad\quad\quad \downarrow_{\widehat{\Psi}(f)} (\widehat{\xi}(f))(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor \ \wedge\ \delta' \in \text{wext}_{\widehat{\Psi}}(\delta''') \\ \quad \text{then } \delta' \in \text{wext}_{\widehat{\Psi}}(\delta), \delta'(\mathtt{res}) = d, \delta'(\mathtt{seen}) = \text{dom}\, d, \text{ and} \\ \quad\quad \forall q \in \delta'(\mathtt{res}).\, \exists f, (v_1, \ldots, v_k), \delta'', \delta'''. \\ \quad\quad\quad 1.\ \downarrow_{\widehat{\Psi}(f)} (\widehat{\xi}(f))(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor \\ \quad\quad\quad 2.\ \delta' \in \text{wext}_{\widehat{\Psi}}(\delta''') \end{array} \right\}.$$

$P$ is clearly pointed; it is also easily verified to be inclusive, noting that any $\omega$-chain $F_i$ will eventually stabilize for each $\delta$ due to $TDecl = [\Delta \to (Decl \times \Delta)_\perp]$ being a function space with a flat co-domain. Note that condition (a.) simply require that the generation of the equations use only new and visit, and condition (b.) is only needed to make the induction go through. The lemma follows by fixed-point induction on $P$, using the operator $\Phi$ underlying $\text{close}_{\widehat{\Psi}}(\widehat{\xi})$. □

For concrete applications, we shall also need to prove that computing the transitive closure actually terminates, if there is a finite bound $X$ for visited functions for each set of given static parameter values (and the generation of residual functions from $X$ does not diverge). To support the Knuth-Morris-Pratt application in Section 3, where we also want to specify size bounds of the generated programs, a simple – although technical – lemma is the most convenient:

**Lemma 4** *If* $\vdash \widehat{d} : \widehat{\Psi}$, $X \subseteq_{\text{fin}} FVar_{\widehat{\Psi}}$, *and* $\forall f_{(v_1,\ldots,v_k)} \in X, \delta.\ (\delta(\texttt{seen}) \subseteq X) \Rightarrow \exists q, \delta'.$

1. $\downarrow_{\widehat{\Psi}(f)} (\widehat{\xi}(f))(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\, \delta = \lfloor (q, \delta') \rfloor \ \wedge\ \delta' \in \text{ext}_{\widehat{\Psi}}(\delta)$
2. $\delta'(\texttt{seen}) \subseteq X$,

*then* $\forall \delta.\, (\delta(\texttt{seen}) \subseteq X \wedge \text{well}_{\widehat{\Psi}}(\delta)) \Rightarrow \exists d, \delta'.\, \text{close}_{\widehat{\Psi}}(\widehat{\xi})\, \delta = \lfloor (d, \delta') \rfloor \ \wedge\ \delta'(\texttt{seen}) \subseteq X.$

**Proof:** Assume that the premise holds, and let $\delta$ be given. The lemma follows by mathematical induction on $|X - \delta(\texttt{res})|$. □

Of course, finding such a finite $X$ is undecidable in general.

### 2.4.2 Symbolic unfolding

Symbolic unfolding turns out to be the main source of complication, for three reasons:

1. Renaming of bound variables had to be included and must be sound (captured as $\lesssim$).
2. The result of unfolding must be correct (captured as $\lessapprox$).
3. Soundness can not be established *equationally* by fixed-point induction (complicating $\lessapprox$).

We will first consider renaming of variables. It will be useful to recall the symbolic unfolding case from the residualizing semantics:

$$[\![\overline{f}(x_1, \ldots, x_n)]\!]_{\mathsf{R}}\, \widehat{\xi}\, \widehat{\rho} = \widehat{\xi}(f)(\widehat{\rho}(x_1), \ldots, \widehat{\rho}(x_n))$$

Here $\widehat{\xi}(f)$ is an expression-generating function. First, we must ensure that $\widehat{\xi}(f)$ is given sound arguments (e.g., to ensure that generated expressions are not untypable).

Given $\widehat{\Gamma}$, $\widehat{\rho} \in [\![\widehat{\Gamma}]\!]_{\mathsf{R}}$, and $\rho \in [\![\|\widehat{\Gamma}\|]\!]$, we define

$$\begin{aligned} \widehat{\rho} \lesssim^i_{\widehat{\Gamma}} \rho \quad \Leftrightarrow \quad & (\forall x \in \text{dom}\,\widehat{\Gamma}.\, S(\widehat{\Gamma}(x)) \Rightarrow \widehat{\rho}(x) = \rho(x)) \wedge \\ & (\forall x \in \text{dom}\,\widehat{\Gamma}.\, D(\widehat{\Gamma}(x)) \Rightarrow \widehat{\rho}(x) \notin \{y_i, y_{i+1}, \ldots\}) \wedge \\ & \forall x, y \in \text{dom}\,\widehat{\Gamma}.\, \widehat{\rho}(x) = \widehat{\rho}(y) \Rightarrow (\widehat{\Gamma}(x) = \widehat{\Gamma}(y) \wedge \rho(x) = \rho(y)) \end{aligned}$$

Thus, $\widehat{\rho} \lesssim^i_{\widehat{\Gamma}} \rho$ asserts that $\widehat{\rho}$ is consistent with $\rho$, in the sense that $\rho$ can be obtained from $\widehat{\rho}$ by assigning values to its variables (which must not clash with name generation).

If $\widehat{\rho} \lesssim^i_{\widehat{\Gamma}} \rho$, let

$$\begin{aligned} \widehat{\Gamma}_\alpha(\widehat{\rho}) &= \{\widehat{\rho}(x) \mapsto \widehat{\Gamma}(x) \,|\, x \in \text{dom}\,\widehat{\Gamma}, D(\widehat{\Gamma}(x))\} \\ \rho^{\widehat{\Gamma}}_\alpha(\widehat{\rho}) &= \{\widehat{\rho}(x) \mapsto \rho(x) \,|\, x \in \text{dom}\,\widehat{\Gamma}, D(\widehat{\Gamma}(x))\} \end{aligned}$$

These functions propagate types and values through the $\alpha$-renaming defined by $\widehat{\rho}$, that is, they describe how typing and value environments are transformed to apply to the generated expression. They are well-defined by $\widehat{\rho}$'s consistency with $\rho$. Note that $\rho^{\widehat{\Gamma}}_\alpha(\widehat{\rho}) \in [\![\|\widehat{\Gamma}_\alpha(\widehat{\rho})\|]\!]$ and that static types and values are not propagated.

We shall need the following simple properties about $\lesssim$.

**Lemma 5** *If* $\widehat{\rho} \lesssim^i_{\widehat{\Gamma}} \rho$ *then*

1. *if* $j > i$ *then* $\widehat{\rho} \lesssim^j_{\widehat{\Gamma}} \rho$.

2. *if* $v \in [\![\tau]\!]$ *then*
   (a) $\widehat{\rho}[x \mapsto v] \lesssim^{i}_{\widehat{\Gamma}[x \mapsto \overline{\tau}]} \rho[x \mapsto v]$
   (b) $\widehat{\Gamma}[x \mapsto \overline{\tau}]_{\alpha}(\widehat{\rho}[x \mapsto v]) = \widehat{\Gamma}_{\alpha}(\widehat{\rho})$
   (c) $\rho[x \mapsto v]^{\widehat{\Gamma}[x \mapsto \overline{\tau}]}_{\alpha}(\widehat{\rho}[x \mapsto v]) = \rho^{\widehat{\Gamma}}_{\alpha}(\widehat{\rho})$

3. *if* $v \in [\![\tau]\!]$ *then*
   (a) $\widehat{\rho}[x \mapsto y_i] \lesssim^{i+1}_{\widehat{\Gamma}[x \mapsto \underline{\tau}]} \rho[x \mapsto v]$
   (b) $\widehat{\Gamma}[x \mapsto \underline{\tau}]_{\alpha}(\widehat{\rho}[x \mapsto y_i]) = (\widehat{\Gamma}_{\alpha}(\widehat{\rho}))[y_i \mapsto \underline{\tau}]$
   (c) $\rho[x \mapsto v]^{\widehat{\Gamma}[x \mapsto \underline{\tau}]}_{\alpha}(\widehat{\rho}[x \mapsto y_i]) = (\rho^{\widehat{\Gamma}}_{\alpha}(\widehat{\rho}))[y_i \mapsto v]$

4. *if* $x \in \operatorname{dom}\widehat{\Gamma}$ *then* $\widehat{\Gamma}_{\alpha}(\widehat{\rho})(\widehat{\rho}(x)) = \widehat{\Gamma}(x)$ *and* $\rho^{\widehat{\Gamma}}_{\alpha}(\widehat{\rho})(\widehat{\rho}(x)) = \rho(x)$.

5. *if* $(\forall j \in [1..n].\, x_j \in \operatorname{dom}\widehat{\Gamma})$ *and* $z_1, \ldots, z_n$ *are distinct variables then*
   (a) $\{z_i \mapsto \widehat{\rho}(x_i) \mid i \in [1..n]\} \lesssim^{i}_{\{z_i \mapsto \widehat{\Gamma}(x_i) \mid i \in [1..n]\}} \{z_i \mapsto \rho(x_i) \mid i \in [1..n]\}$
   (b) $\{z_i \mapsto \widehat{\Gamma}(x_i) \mid i \in [1..n]\}_{\alpha}(\{z_i \mapsto \widehat{\rho}(x_i) \mid i \in [1..n]\}) = \widehat{\Gamma}_{\alpha}(\widehat{\rho})$
   (c) $\{z_i \mapsto \rho(x_i) \mid i \in [1..n]\}^{\{z_i \mapsto \widehat{\Gamma}(x_i) \mid i \in [1..n]\}}_{\alpha}(\{z_i \mapsto \widehat{\rho}(x_i) \mid i \in [1..n]\}) = \rho^{\widehat{\Gamma}}_{\alpha}(\widehat{\rho})$

**Proof:** Straightforward in all cases. □

We now consider the two last reasons. We first illustrate the latter, technical reason: recall the power function from the introduction (glossing over syntactic restrictions),

$$\widehat{q} = \texttt{power(}n\texttt{,x) = }\textit{if (n=0) then }\texttt{1}\textit{ else }\texttt{x*(}\textit{power(n-1,}\texttt{x)}\textit{)}$$

(placed in context $\widehat{p} = \texttt{local [}\widehat{q}\texttt{] in power}$) and its specialized version,

$$q = \texttt{power}_2\texttt{(x) = x*(x*1)}$$

(placed in context $p = \texttt{local [}q\texttt{] in power}_2$). We thus have $[\![\widehat{p}]\!]_{\mathsf{R}}(2) = \lfloor p \rfloor$. By soundness of specialization, the following equality is required to hold (unfolding the definition for curry):

$$[\![p]\!] = \lambda x.\, [\![\|\widehat{p}\|]\!](2, x)$$

By further unfolding the definitions, we can recognize it as a relation between two fixed points,

$$\operatorname{fix}(\Phi')(\texttt{power}_2) = \lambda x.\, \operatorname{fix}(\Phi)(\texttt{power})(2, x)$$

where $\Phi' = \lambda \xi'.\, \{\texttt{power}_2 \mapsto [\![q]\!]\, \xi'\}$ and $\Phi = \lambda \xi.\, \{\texttt{power} \mapsto [\![\|\widehat{q}\|]\!]\, \xi\}$. The standard technique for proving such equalities is fixed-point induction, here on the pointed and inclusive relation $P = \{(\xi', \xi) \mid \xi'(\texttt{power}_2) = \lambda x.\, \xi(\texttt{power})(2, x)\}$. But we cannot prove the inductive case, i.e., if $(\xi', \xi) \in P$ then $(\Phi'(\xi'), \Phi'(\xi)) \in P$, because it does not hold. The problem can be illustrated using the example, where we have by definition (simplifying the total arithmetic functions):

$$\begin{aligned} [\![\|\widehat{q}\|]\!]\, \xi &= \lambda(n, x).\, \text{case } (n = 0) \text{ of } \begin{cases} \mathrm{tt} \to 1 \\ \mathrm{ff} \to \xi(\texttt{power})(n - 1, x) \dagger \lambda r.\, \lfloor x \times r \rfloor \end{cases} \\ [\![q]\!]\, \xi' &= \lambda x.\, \lfloor x \times x \rfloor \end{aligned}$$

Then,

$$\begin{array}{rccc} \lambda x.\, \bot(\texttt{power})(2, x) = & \bot & = & \bot & = \bot(\texttt{power}_2) \\ \lambda x.\, \Phi(\bot)(\texttt{power})(2, x) = & \bot & \neq \lambda x.\, \lfloor x \times x \rfloor & = \Phi'(\bot)(\texttt{power}_2) \\ \lambda x.\, \Phi(\Phi(\bot))(\texttt{power})(2, x) = & \bot & \neq \lambda x.\, \lfloor x \times x \rfloor & = \Phi'(\Phi'(\bot))(\texttt{power}_2) \\ \lambda x.\, \Phi(\Phi(\Phi(\bot)))(\texttt{power})(2, x) = & \lambda x.\, \lfloor x \times x \rfloor & = \lambda x.\, \lfloor x \times x \rfloor & = \Phi'(\Phi'(\Phi'(\bot)))(\texttt{power}_2) \\ & \text{etc.} & & \end{array}$$

So although the desired equality certainly does hold, we cannot in general prove it by fixed-point induction alone. We can observe that fixed-point induction may give us one direction, namely

$$\lambda x.\, \mathrm{fix}(\Phi)(\mathtt{power})(2,x) \sqsubseteq \mathrm{fix}(\Phi')(\mathtt{power}_2)$$

This direction does actually hold in general.

For the other direction, consider the residual meaning of power:

$$[\![\widehat{q}]\!]_{\mathsf{R}}\,\widehat{\xi} = \lambda(n,x).\,\mathrm{case}\ (n=0)\ \mathrm{of}\ \begin{cases} \mathtt{tt} \to \mathtt{1} \\ \mathtt{ff} \to \widehat{\xi}(\mathtt{power})(n-1,x) \dagger \lambda r.\, \lfloor x \mathtt{*} r \rfloor \end{cases}$$

Let $\tilde{\Phi} = \widehat{\xi}.\{\mathtt{power} \mapsto [\![\widehat{q}]\!]_{\mathsf{R}}\,\widehat{\xi}\}$. Here, the symbolic unfoldings are

$$\begin{array}{rll} \bot(2,\mathtt{x}) = \bot & & = \bot \\ \tilde{\Phi}(\bot)(2,\mathtt{x}) = \bot(1,\mathtt{x}) \dagger \lambda r.\, \lfloor \mathtt{x*}r \rfloor & & = \bot \\ \tilde{\Phi}(\tilde{\Phi}(\bot))(2,\mathtt{x}) = \bot(0,\mathtt{x}) \dagger \lambda r.\, \lfloor \mathtt{x*}r \rfloor \dagger \lambda r.\, \lfloor \mathtt{x*}r \rfloor & & = \bot \\ \tilde{\Phi}(\tilde{\Phi}(\tilde{\Phi}(\bot)))(2,\mathtt{x}) = \lfloor \mathtt{1} \rfloor \dagger \lambda r.\, \lfloor \mathtt{x*}r \rfloor \dagger \lambda r.\, \lfloor \mathtt{x*}r \rfloor & & = \lfloor \mathtt{x*x*1} \rfloor \\ \multicolumn{3}{c}{\text{etc.}} \end{array}$$

The fallacy of the above example was to assume that the specialization terminated, thereby implicitly using $\mathrm{fix}(\tilde{\Phi})$ to define the operator $\Phi'$ and thus unavoidably baking adequate symbolic unfolding into it. So while $\Phi^i(\bot)$ and $\tilde{\Phi}^i(\bot)$ are in perfect harmony, $\Phi^i(\bot)$ and $\Phi'^i(\bot)$ are not until the regular unfolding has "caught up" with the symbolic unfolding (which happens at $i=3$ in the example). A central concept in the proof is precisely this relationship, written $\lessapprox$.

Given $\widehat{\Psi}$, $\widehat{\xi} \in [\![\widehat{\Psi}]\!]_{\mathsf{R}}$, and $\xi \in [\![\|\widehat{\Psi}\|]\!]$, we define

$$\begin{array}{rl} \widehat{\xi} \lessapprox_{\widehat{\Psi}} \xi \ \Leftrightarrow\ & (\forall f \in \mathrm{dom}\,\widehat{\Psi}.\, S(\widehat{\Psi}(f)) \Rightarrow \widehat{\xi}(f) = \gamma_T \circ \xi(f)) \wedge \\ & \forall f \in \mathrm{dom}\,\widehat{\Psi}.D(\widehat{\Psi}(f)) \Rightarrow \forall \widehat{\tau}_1, \ldots, \widehat{\tau}_n, \tau, \widehat{\Gamma}, x_1, \ldots, x_n, \widehat{\rho}, \rho, \delta, \delta', \xi', e. \\ & \text{if } \widehat{\Psi}(f) = \widehat{\tau}_1 \mathtt{*} \ldots \mathtt{*} \widehat{\tau}_n \mathtt{->} \underline{\tau}, \mathrm{dom}\,\widehat{\Gamma} = \{x_1, \ldots, x_n\}, (\forall i \in [1..n].\, \widehat{\Gamma}(x_i) = \widehat{\tau}_i), \\ & \quad \widehat{\rho} \lessapprox_{\widehat{\Gamma}}^{\delta(\mathtt{i})} \rho,\ \xi' \sqsubseteq \xi, \text{ and } \widehat{\xi}(f)(\widehat{\rho}(x_1), \ldots, \widehat{\rho}(x_n))\,\delta = \lfloor (e, \delta') \rfloor \\ & \text{then } \delta' \in \mathrm{ext}_{\widehat{\Psi}}(\delta) \text{ and, letting } X = \delta'(\mathtt{seen}), \\ & \quad 1.\ \|\underline{\widehat{\Psi}}(X)\|, \|\widehat{\Gamma}_\alpha(\widehat{\rho})\| \vdash e : \tau \\ & \quad 2.\ [\![e]\!]\,(\mathrm{curry}_{\widehat{\Psi}}(\xi)(X))\,\rho_\alpha^{\widehat{\Gamma}}(\widehat{\rho}) = \xi(f)(\rho(x_1), \ldots, \rho(x_n)) \\ & \quad 3.\ \xi'(f)(\rho(x_1), \ldots, \rho(x_n)) \sqsubseteq [\![e]\!]\,(\mathrm{curry}_{\widehat{\Psi}}(\xi')(X))\,\rho_\alpha^{\widehat{\Gamma}}(\widehat{\rho}) \end{array}$$

There are four parts (beyond bookkeeping): the static case and the numbered dynamic conclusions.

The static case (the first line) simply requires that $\widehat{\xi}(f)$ is identical to $\xi(f)$ (modulo $\eta$).

The dynamic case (the remaining lines) addresses several issues, given the premise that the symbolic unfolding of $\widehat{\xi}(f)$ terminates and is called from some valid typing context $\widehat{\Gamma}$ with sound renaming $\widehat{\rho} \lessapprox_{\widehat{\Gamma}} \rho$. Then, (case 1.) the generated expression must be well-typed in the residual context; (case 2.) the meaning of it in this context must be equal to the meaning of the erasured function call; and (case 3.) if further $\xi' \sqsubseteq \xi$ then only one direction of (case 2.) is guaranteed to hold when substituting $\xi'$ for $\xi$. We can thus recognize the two cases of symbolic unfolding mentioned above: (case 2.) can be read as $\xi$ is "adequately unfolded" with repect to $\widehat{\xi}$, and (case 3.) as any $\xi' \sqsubseteq \xi$ is perhaps "inadequately unfolded" with repect to $\widehat{\xi}$.

We can now present an outline for the soundness proof (illustrated by power).

1. Show by structural induction that, given $\widehat{\xi} \lessapprox \xi$, the RHS generalizes to all terms, not just the case for symbolic unfolding (Lemmas 7-10).
2. Establish $[\![[\widehat{q}]]\!]_{\mathsf{R}} \lessapprox [\![\|[\widehat{q}]\|]\!]$, i.e., $\mathrm{fix}(\tilde{\Phi}) \lessapprox \mathrm{fix}(\Phi)$, by fixed-point induction (Lemma 11).

3. Use (case 3.) of the RHS, taking $\Phi^i(\bot) \sqsubseteq \mathrm{fix}(\Phi)$, to show

$$\lambda x.\, \mathrm{fix}(\Phi)(\mathtt{power})(2,x) \sqsubseteq \mathrm{fix}(\Phi')(\mathtt{power}_2)$$

by fixed-point induction. Further,

$$\begin{array}{ll}
\Phi'(\{\mathtt{power}_2 \mapsto \lambda x.\, \mathrm{fix}(\Phi)(\mathtt{power})(2,x)\}) & \\
= \{\mathtt{power}_2 \mapsto [\![q]\!]\,\{\mathtt{power}_2 \mapsto \lambda x.\, \mathrm{fix}(\Phi)(\mathtt{power})(2,x)\}\} & \\
= \{\mathtt{power}_2 \mapsto \lambda x.\, ([\![\|\widehat{q}\|]\!]\,\mathrm{fix}(\Phi))(2,x)\} & \text{By (case 2.) of the RHS} \\
= \{\mathtt{power}_2 \mapsto \lambda x.\, (\Phi(\mathrm{fix}(\Phi))(\mathtt{power}))(2,x)\} & \\
= \{\mathtt{power}_2 \mapsto \lambda x.\, \mathrm{fix}(\Phi)(\mathtt{power})(2,x)\} & \text{By } \Phi(\mathrm{fix}(\Phi)) = \mathrm{fix}(\Phi)
\end{array}$$

Hence, $\{\mathtt{power}_2 \mapsto \lambda x.\, \mathrm{fix}(\Phi)(\mathtt{power})(2,x)\}$ is a fixed point of $\Phi'$ and therefore greater than its *least* fixed point, $\mathrm{fix}(\Phi')$. Consequently,

$$\lambda x.\, \mathrm{fix}(\Phi)(\mathtt{power})(2,x) \sqsupseteq \mathrm{fix}(\Phi')(\mathtt{power}_2)$$

This completes the key lemma (Lemma 12), which implies soundness (Theorem 1).

The typing parts are unproblematic, so is accounting for the transitive closure.

We have a technical requirement for $\lessapprox$, namely that is admits fixed-point induction:

**Lemma 6** *For fixed $\widehat{\Psi}$, $P = \{(\widehat{\xi},\xi) \mid \widehat{\xi} \lessapprox_{\widehat{\Psi}} \xi\}$ is pointed (i.e., $(\bot,\bot) \in P$) and inclusive (i.e., closed under $\omega$-chains).*

**Proof:** Straightforward verification. □

**Remark 1** *It may be instructive to consider omitting symbolic unfolding. There would be no need for renaming, so dynamic variables could be syntactically preserved by the residualizing semantics (the restriction that function arguments must be variables also becomes unnecessary). The definitions of this section are considerably simplified:*

$$\begin{array}{rl}
\widehat{\rho} \lesssim^i_{\widehat{\Gamma}} \rho \Leftrightarrow & \forall x \in \mathrm{dom}\,\widehat{\Gamma}.\, S(\widehat{\Gamma}(x)) \Rightarrow \widehat{\rho}(x) = \rho(x) \\
\widehat{\Gamma}_\alpha(\widehat{\rho}) = & \{x \mapsto \widehat{\Gamma}(x) \mid x \in \mathrm{dom}\,\widehat{\Gamma}, D(\widehat{\Gamma}(x))\} \\
\rho^{\widehat{\Gamma}}_\alpha(\widehat{\rho}) = & \{x \mapsto \rho(x) \mid x \in \mathrm{dom}\,\widehat{\Gamma}, D(\widehat{\Gamma}(x))\} \\
\widehat{\xi} \lessapprox_{\widehat{\Psi}} \xi \Leftrightarrow & \forall f \in \mathrm{dom}\,\widehat{\Psi}.\, S(\widehat{\Psi}(x)) \Rightarrow \widehat{\xi}(f) = \gamma_T \circ \xi(f)
\end{array}$$

*Most importantly, the simple fixed-point induction is now strong enough to prove correctness.*

### 2.4.3 Soundness of the residualizing semantics

Soundness of the residualizing semantics follows the proof outline from above.

Static terms form a proper subset of the language, where the non-standard semantics agrees with the standard ones:

**Lemma 7** *If $\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e} : \overline{\tau}$, $\widehat{\xi} \lessapprox_{\widehat{\Psi}} \xi$, and $\widehat{\rho} \lesssim^i_{\widehat{\Gamma}} \rho$ then $[\![\widehat{e}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho} = \gamma_T([\![\|\widehat{e}\|]\!]\,\xi\,\rho)$.*

**Proof:** Straightforward, by structural induction. □

**Lemma 8** *If $\widehat{\Psi} \vdash \widehat{q} : (f,\widehat{\sigma})$, $S(\widehat{\sigma})$, and $\widehat{\xi} \lessapprox_{\widehat{\Psi}} \xi$ then $[\![\widehat{q}]\!]_{\mathsf{R}}\,\widehat{\xi} = \gamma_T \circ ([\![\|\widehat{q}\|]\!]\,\xi)$.*

**Proof:** Follows directly from lemma 7. □

Dynamic terms and their residualized versions have the same meaning in the standard semantics, taking appropriate contexts into account.

**Lemma 9** *If $\widehat{\Psi},\widehat{\Gamma} \vdash \widehat{e} : \underline{\tau}$, $\widehat{\xi} \lessapprox_{\widehat{\Psi}} \xi$, $\widehat{\rho} \lesssim^{\delta(\mathtt{i})}_{\widehat{\Gamma}} \rho$, $\xi' \sqsubseteq \xi$, and $([\![\widehat{e}]\!]_{\mathsf{R}}\,\widehat{\xi}\,\widehat{\rho})\,\delta = \lfloor (e,\delta') \rfloor$ then $\delta' \in \mathrm{ext}_{\widehat{\Psi}}(\delta)$ and, letting $X = \delta'(\mathtt{seen})$,*

1. $\|\underline{\widehat{\Psi}}(X)\|, \|\widehat{\Gamma}_\alpha(\widehat{\rho})\| \vdash e : \tau$
2. $[\![e]\!]\,(\mathrm{curry}_{\widehat{\Psi}}(\xi)(X))\,\rho_\alpha^{\widehat{\Gamma}}(\widehat{\rho}) = [\![\|\widehat{e}\|]\!]\,\xi\,\rho$
3. $[\![\|\widehat{e}\|]\!]\,\xi'\,\rho \sqsubseteq [\![e]\!]\,(\mathrm{curry}_{\widehat{\Psi}}(\xi')(X))\,\rho_\alpha^{\widehat{\Gamma}}(\widehat{\rho})$

**Proof:** By structural induction, using Lemmas 1, 5(1-4), and 7. □

**Lemma 10** *If* $\widehat{\Psi} \vdash \widehat{q} : (f, \widehat{\sigma})$, $D(\widehat{\sigma})$, $\widehat{\xi} \lessapprox_{\widehat{\Psi}} \xi$, $\xi' \sqsubseteq \xi$, $(v_1, \ldots, v_k) \in [\![\|\overline{\widehat{\sigma}}\|]\!]$, *and furthermore* $(\downarrow_{\widehat{\sigma}} ([\![\widehat{q}]\!]_{\mathsf{R}}\,\widehat{\xi})(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k)))\,\delta = \lfloor (q, \delta') \rfloor$ *then* $\delta' \in \mathrm{ext}_{\widehat{\Psi}}(\delta)$ *and, letting* $X = \delta'(\mathtt{seen})$,

1. $\|\underline{\widehat{\Psi}}(X)\| \vdash q : (f_{(v_1,\ldots,v_k)}, \|\underline{\widehat{\sigma}}\|)$
2. $[\![q]\!]\,(\mathrm{curry}_{\widehat{\Psi}}(\xi)(X)) = \mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{q}\|]\!]\,\xi)(v_1, \ldots, v_k)$
3. $\mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{q}\|]\!]\,\xi')(v_1, \ldots, v_k) \sqsubseteq [\![q]\!]\,(\mathrm{curry}_{\widehat{\Psi}}(\xi')(X))$

**Proof:** Follows from Lemma 9, using Lemma 5(1-3). □

The declaration level is where all the pieces must fit together. First, we need to establish that a residualized declaration provides sound symbolic unfolding:

**Lemma 11** *If* $\vdash \widehat{d} : \widehat{\Psi}$ *then* $[\![\widehat{d}]\!]_{\mathsf{R}} \lessapprox_{\widehat{\Psi}} [\![\|\widehat{d}\|]\!]$.

**Proof:** By fixed-point induction, using Lemmas 5(5), 6, 8, and 10. □

We are now in position to establish the key soundness lemma.

**Lemma 12** *If* $\vdash \widehat{d} : \widehat{\Psi}$ *and* $(\mathrm{visit}(f) \star \lambda u.\, \mathrm{close}_{\widehat{\Psi}}([\![\widehat{d}]\!]_{\mathsf{R}}))\,\delta_0 = \lfloor (d, \delta) \rfloor$ *then, letting* $X = \delta(\mathtt{seen})$,

1. $\vdash d : \|\underline{\widehat{\Psi}}(X)\|$
2. $[\![d]\!] = \mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X)$

**Proof:** Assume that $\vdash \widehat{d} : \widehat{\Psi}$ and $(\mathrm{visit}(f) \star \lambda u.\, \mathrm{close}_{\widehat{\Psi}}([\![\widehat{d}]\!]_{\mathsf{R}}))\,\delta_0 = \lfloor (d, \delta) \rfloor$. Note first the conditions for Lemma 3 are straightforwardly satisfied, using the simple extension part of Lemma 10 (formally triggered by Lemma 11).

(1.) We aim to establish that each equation $q \in d$ is well-typed in typing context $\|\underline{\widehat{\Psi}}(X)\|$ and that $\mathrm{dom}\, d = \mathrm{dom}\, \|\underline{\widehat{\Psi}}(X)\| = X$, i.e., that the premise for the typing rule for declarations holds.

Let $q \in d$ be arbitrary. By Lemma 3(1), there is some $f' \in \mathrm{dom}\, \widehat{\Psi}$, $(v_1, \ldots, v_k)$, $\delta''$, and $\delta'''$ such that $\downarrow_{\widehat{\Psi}(f')} ([\![\widehat{d}]\!]_{\mathsf{R}}(f'))(f'_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor$. Let $Y = \delta'''(\mathtt{seen})$. By Lemma 10(1) (formally triggered by Lemma 11),

$$\|\underline{\widehat{\Psi}}(Y)\| \vdash q : (f'_{(v_1,\ldots,v_k)}, \underline{\widehat{\sigma}})$$

By Lemma 3(2), $Y \subseteq X$, so by weakening (Lemma 2(1)),

$$\|\underline{\widehat{\Psi}}(X)\| \vdash q : (f'_{(v_1,\ldots,v_k)}, \underline{\widehat{\sigma}})$$

Since $q$ was arbitrary, each $q \in d$ is well-typed in context $\|\underline{\widehat{\Psi}}(X)\|$.

By Lemma 3, $\delta'(\mathtt{seen}) = \mathrm{dom}\, d$, i.e., $X = \mathrm{dom}\, d$.

The conclusion, $\vdash d : \|\underline{\widehat{\Psi}}(X)\|$, then follows by the typing rule for declarations.

(2.) Let $\Phi$ and $\Phi'$ denote the operators underlying $[\![\|\widehat{d}\|]\!]$ and $[\![d]\!]$, resp. We consider each direction of the equality separately, as motivated in Section 2.4.2.

**Case** $\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X) \sqsubseteq [\![d]\!]$: Let

$$P = \{(\xi, \xi') \in [\![\|\widehat{\Psi}\|]\!] \times [\![\|\underline{\widehat{\Psi}}(X)\|]\!] \mid \mathrm{curry}_{\widehat{\Psi}}(\xi)(X) \sqsubseteq \xi' \ \wedge \ \xi \sqsubseteq [\![\|\widehat{d}\|]\!]\}$$

It is straightforward to verify that the predicate is pointed and inclusive. We proceed by fixed-point induction. Assume $(\xi, \xi') \in P$.

Let $f'_{(v_1,\ldots,v_k)} \in X$ be arbitrary. By well-typedness of $\widehat{d}$, there is some $\widehat{q} \in \widehat{d}$ and $\widehat{\sigma}$ such that $\widehat{\Psi} \vdash \widehat{q} : (f', \widehat{\sigma})$. By Lemma 3(1) and part 1, there exists $q \in d, \delta''$ and $\delta'''$ such that $\downarrow_{\widehat{\sigma}}([\![\widehat{d}]\!]_{\mathsf{R}}(f'))(f'_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor$. Let $Y = \delta'''(\mathtt{seen})$. Now,

$$\begin{array}{ll}
& \mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{q}\|]\!]\,\xi)(v_1, \ldots, v_k) \\
\sqsubseteq & [\![q]\!]\,\mathrm{curry}_{\widehat{\Psi}}(\xi)(Y) \qquad \text{By } \xi \sqsubseteq [\![\|\widehat{d}\|]\!] \text{ and Lemmas 10(3) and 11} \\
= & [\![q]\!]\,\mathrm{curry}_{\widehat{\Psi}}(\xi)(X) \qquad \text{By Lemmas 3(2) and 2(2)} \\
\sqsubseteq & [\![q]\!]\,\xi' \qquad \text{By IH and monotonicity}
\end{array}$$

Since $f_{(v_1,\ldots,v_k)} \in X$ was arbitrary,

$$\begin{array}{l}
\mathrm{curry}_{\widehat{\Psi}}(\Phi(\xi))(X) \\
= \{f_{(v_1,\ldots,v_k)} \mapsto \mathrm{curry}_{\widehat{\sigma}}(\Phi(\xi)(f))(v_1, \ldots, v_k) \mid f_{(v_1,\ldots,v_k)} \in X\} \\
= \{f_{(v_1,\ldots,v_k)} \mapsto \mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{q}\|]\!]\,\xi)(v_1, \ldots, v_k) \mid f_{(v_1,\ldots,v_k)} \in X\} \\
\sqsubseteq \{f_{(v_1,\ldots,v_k)} \mapsto [\![q]\!]\,\xi' \mid f_{(v_1,\ldots,v_k)} \in X\} \\
= \Phi'(\xi')
\end{array}$$

By monotonicity of $\Phi$ and the fixed-point equation, $\Phi(\xi) \sqsubseteq \Phi([\![\|\widehat{d}\|]\!]) = [\![\|\widehat{d}\|]\!]$. Hence, $(\Phi(\xi), \Phi'(\xi')) \in P$; we can thus conclude $\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X) \sqsubseteq [\![d]\!]$.

**Case** $[\![d]\!] \sqsubseteq \mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X)$: The idea is to show that $\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X)$ is a fixed point of $\Phi'$ and hence greater than its least fixed point, $[\![d]\!]$.

Let $f'_{(v_1,\ldots,v_k)} \in X$ be arbitrary. By well-typedness of $\widehat{d}$, there is some $\widehat{q} \in \widehat{d}$ and $\widehat{\sigma}$ such that $\widehat{\Psi} \vdash \widehat{q} : (f', \widehat{\sigma})$. By Lemma 3(1) and part 1, there exists $q \in d, \delta''$ and $\delta'''$ such that $\downarrow_{\widehat{\sigma}}([\![\widehat{d}]\!]_{\mathsf{R}}(f'))(f'_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k))\ \delta'' = \lfloor (q, \delta''') \rfloor$. Let $Y = \delta'''(\mathtt{seen})$. Thus,

$$\begin{array}{ll}
& \mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{q}\|]\!]\ [\![\|\widehat{d}\|]\!])(v_1, \ldots, v_k) \\
= & [\![q]\!]\,\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(Y) \qquad \text{By Lemmas 10(2) and 11} \\
= & [\![q]\!]\,\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X) \qquad \text{By Lemmas 3(2) and 2(2)}
\end{array}$$

Since $f_{(v_1,\ldots,v_k)} \in X$ was arbitrary,

$$\begin{array}{l}
\Phi'(\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X)) \\
= \{f_{(v_1,\ldots,v_k)} \mapsto [\![q]\!]\,\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X) \mid f_{(v_1,\ldots,v_k)} \in X\} \\
= \{f_{(v_1,\ldots,v_k)} \mapsto \mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{q}\|]\!]\ [\![\|\widehat{d}\|]\!])(v_1, \ldots, v_k) \mid f_{(v_1,\ldots,v_k)} \in X\} \\
= \mathrm{curry}_{\widehat{\Psi}}(\Phi([\![\|\widehat{d}\|]\!]))(X) \\
= \mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X) \qquad \text{Since } [\![\|\widehat{d}\|]\!] \text{ is a fixed point of } \Phi
\end{array}$$

Hence, $\mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X)$ is a fixed point of $\Phi'$. Consequently,

$$[\![d]\!] = \mathrm{fix}(\Phi') \sqsubseteq \mathrm{curry}_{\widehat{\Psi}}([\![\|\widehat{d}\|]\!])(X)$$

□

**Theorem 1 (soundness)** *If* $\vdash \widehat{p} : \widehat{\sigma}$ *and* $[\![\widehat{p}]\!]_{\mathsf{R}}(v_1, \ldots, v_k) = \lfloor p \rfloor$ *then*

1. $\vdash p : \underline{\widehat{\sigma}}$
2. $[\![p]\!] = \mathrm{curry}_{\widehat{\sigma}}([\![\|\widehat{p}\|]\!])(v_1, \ldots, v_k)$

**Proof:** Follows directly from Lemma 12, using Lemma 3. □

# 3 Application: the Knuth-Morris-Pratt algorithm

String matching has been a subject to extensive investigation in partial evaluation, since Futamura in 1987 presented the method of Generalized Partial Computation [26]. His method could transform a naive, inefficient string matcher into the search phase of the ingenious Knuth-Morris-Pratt algorithm [35]. This example (now known as the "KMP-test" [41]) has since fueled numerous investigations in partial evaluation [1–4, 12, 29, 39–41]. In particular, Consel and Danvy showed that polyvariant specialization is acutally strong enough to pass the KMP-test, for certain inefficient string matchers [12]. One notable utility of the KMP-test has been to relate and distinguish various advanced program transformations with respect to how much information is propagated (and exploited) by the transformations [39, 41].

However, only recently has it been shown precisely what information should be propagated to obtain the Knuth-Morris-Pratt algorithm [1]. As an amusing corollary, none of the advanced transformations that pass the KMP-test actually propagate the precise amount of information needed to exactly obtain the Knuth-Morris-Pratt algorithm [39].

The value of the formal treatment was thus to pinpoint this precise amount of information. For the purpose of analysis, a simple polyvariant specializer that actually failed the KMP-test was used [1]. The reason being that it is much easier to control the source program, which propagates collected static information about the dynamic argument by itself — a so-called binding-time-improved algorithm [12], than an advanced transformation.

From a semantic perspective, however, one piece of the formal treatment is still missing, namely a proof that applying a polyvariant specializer always gives a correct result. We are now in position to address this problem in full.

We instantiate the signature, $\Sigma = (\mathrm{B}, \mathrm{L}, \mathrm{C})$, as follows:

$$
\begin{aligned}
\mathrm{B} &= \{\texttt{int}, \texttt{char}, \texttt{string}\} \\
\mathrm{L}(\texttt{int}) &= \{\ldots, \texttt{-1}, \texttt{0}, \texttt{1}, \ldots\} \\
\mathrm{L}(\texttt{char}) &= \{\texttt{a}, \ldots, \texttt{z}\} \\
\mathrm{L}(\texttt{string}) &= \{\texttt{"}c_1 \cdots c_n\texttt{"} \mid n \in \mathbb{N}, \forall i \in [1..n].\, c_i \in \mathrm{L}(\texttt{char})\} \\
\mathrm{C} &= \{\texttt{+} \mapsto \texttt{int*int->int}, \texttt{-} \mapsto \texttt{int*int->int}, \\
&\qquad \texttt{=}_{\texttt{int}} \mapsto \texttt{int*int->bool}, \texttt{=}_{\texttt{char}} \mapsto \texttt{char*char->bool}, \\
&\qquad \texttt{ref} \mapsto \texttt{string*int->char}, \texttt{length} \mapsto \texttt{string->int}\}
\end{aligned}
$$

The interpretation of the signature is the obvious one.

For conciseness and readability, we (1) abbreviate both $\texttt{=}_{\texttt{int}}$ and $\texttt{=}_{\texttt{char}}$ as =; (2) write binary function constants in infix notation; (3) assume implicit let-insertion of function arguments in a left-to-right order; and (4) write all static, reducible types and terms in italics, instead of overlining them.

We use the following inefficient string matcher, *STAGED* [1].

```
local
 main    : string*string->int (pat,txt)
         = match(pat,txt,0,$0,length(pat),length(txt))
 match   : string*string*int*int*int*int->int (pat,txt,j,k,lp,lt)
         = if (j=lp)
           then (k-$j)
           else if (k=lt)
                then $-1
                else compare(pat,txt,j,k,lp,lt)
 compare : string*string*int*int*int*int->int (pat,txt,j,k,lp,lt)
         = if ($(ref(pat,j))=ref(txt,k))
           then match(pat,txt,j+1,k+$1,lp,lt)
           else if (j=0)
                then match(pat,txt,0,k+$1,lp,lt)
                else rematch(pat,txt,j,k,0,1,lp,lt)
```

```
rematch : string*string*int*int*int*int*int->int
          (pat,txt,j,k,jp,kp,lp,lt)
        = if (kp=j)
          then if (ref(pat,jp)=ref(pat,kp))
               then if (jp=0)
                    then match(pat,txt,0,k+$1,lp,lt)
                    else rematch(pat,txt,j,k,0,(kp-jp)+1,lp,lt)
               else compare(pat,txt,jp,k,lp,lt)
          else if (ref(pat,jp)=ref(pat,kp))
               then rematch(pat,txt,j,k,jp+1,kp+1,lp,lt)
               else rematch(pat,txt,j,k,0,(kp-jp)+1,lp,lt)
in main
```

The expensive part of the computation –called *backtracking*– is encapsulated in the essentially static `rematch` function. During specialization, `rematch` is thus completely unfolded and the residual program runs in linear time (with a constant factor independent of the pattern `pat` [41]). So although inefficient, the string matcher is not particularly naive.

In contrast, more advanced program transformations such as Supercompilation [28, 41, 42] and Generalized Partial Computation [25–27] can pass the KMP-test from a naive (i.e., not staged), inefficient string matcher.

As an example, the result of specializing the inefficient string matcher with repect to the pattern `"abaa"` is shown below (we use the notational conventions from above and have further renamed all variables):

```
local
  main("abaa")        : string->int (txt)
                        = match("abaa",0,4)(txt,0,length(txt))
  match("abaa",0,4)   : string*int*int->int (txt,k,lt)
                        = if k=lt then -1 else compare("abaa",0,4)(txt,k,lt)
  compare("abaa",0,4) : string*int*int->int (txt,k,lt)
                        = if 'a'=ref(txt,k) then match("abaa",1,4)(txt,k+1,lt)
                                            else match("abaa",0,4)(txt,k+1,lt)
  match("abaa",1,4)   : string*int*int->int (txt,k,lt)
                        = if k=lt then -1 else compare("abaa",1,4)(txt,k,lt)
  compare("abaa",1,4) : string*int*int->int (txt,k,lt)
                        = if 'b'=ref(txt,k) then match("abaa",2,4)(txt,k+1,lt)
                                            else compare("abaa",0,4)(txt,k,lt)
  match("abaa",2,4)   : string*int*int->int (txt,k,lt)
                        = if k=lt then -1 else compare("abaa",2,4)(txt,k,lt)
  compare("abaa",2,4) : string*int*int->int (txt,k,lt)
                        = if 'a'=ref(txt,k) then match("abaa",3,4)(txt,k+1,lt)
                                            else match("abaa",0,4)(txt,k+1,lt)
  match("abaa",3,4)   : string*int*int->int (txt,k,lt)
                        = if k=lt then -1 else compare("abaa",3,4)(txt,k,lt)
  compare("abaa",3,4) : string*int*int->int (txt,k,lt)
                        = if 'a'=ref(txt,k) then match("abaa",4,4)(txt,k+1,lt)
                                            else compare("abaa",1,4)(txt,k,lt)
  match("abaa",4,4)   : string*int*int->int (txt,k,lt)
                        = k-4
in main("abaa")
```

We will now show that for all patterns, our polyvariant specializer transforms the inefficient string matcher, *STAGED*, into a program with the above structure. The proof consists of two parts: termination of the specializer and properties of the residual programs.

We define the size of a term, $|\cdot|$, in the obvious way by induction.

**Theorem 2** *For all strings $s$ there exists a $p$ such that $[\![STAGED]\!]_{\mathsf{R}}\, s = \lfloor p \rfloor$ and*

1. $\vdash p$ : `string->int`
2. $[\![p]\!] = \mathrm{curry}_{\mathit{string}*\underline{\texttt{string}}\texttt{->}\underline{\texttt{int}}}([\![\|\mathit{STAGED}\|]\!])(s)$
3. $|p| \in O(|s|)$

**Proof:** Let $s$ be arbitrary. We will establish the prerequisites for Lemma 4. Define

$$X = \{\texttt{main}_{(s)}\} \cup \{\texttt{match}_{(s,j,|s|)} \mid j \in [0..|s|]\} \cup \{\texttt{compare}_{(s,j,|s|)} \mid j \in [0..|s|-1]\}$$

Clearly $|X| = 1 + (|s|+1) + |s|$. Let $\delta$ be given and let $Z = \delta(\texttt{seen}) \subseteq X$. Let $\widehat{\sigma} = \mathit{string}*\underline{\texttt{string}}*\mathit{int}*\underline{\texttt{int}}*\mathit{int}*\underline{\texttt{int}}\texttt{->}\underline{\texttt{int}}$.

Let $f \in X$ be arbitrary. All cases but the last are completely straightforward.

**case** $f = \texttt{main}_{(s)}$: Let $\widehat{\sigma}' = \mathit{string}*\underline{\texttt{string}}\texttt{->}\underline{\texttt{int}}$. Then

$$(\downarrow_{\widehat{\sigma}'} ([\![\mathit{STAGED}]\!]_{\mathsf{R}}(\texttt{main}))(\texttt{main}_{(s)}, (s)))\,\delta = \lfloor (q, \delta') \rfloor$$

where $\delta'(\texttt{seen}) = Z \cup \{\texttt{match}_{(s,0,|s|)}\}$
and $q = \texttt{main}_{(s)}\texttt{: } \underline{\widehat{\sigma}'}\ \texttt{(}y_i\texttt{) = match}_{(s,0,|s|)}\texttt{(}y_i\texttt{,0,length(}y_i\texttt{))}.$

**case** $f = \texttt{match}_{(s,j,|s|)}, j \in [0..|s|-1]$: Then

$$(\downarrow_{\widehat{\sigma}} ([\![\mathit{STAGED}]\!]_{\mathsf{R}}(\texttt{match}))(\texttt{match}_{(s,j,|s|)}, (s, j, |s|)))\,\delta = \lfloor (q, \delta') \rfloor$$

where $\delta'(\texttt{seen}) = Z \cup \{\texttt{compare}_{(s,j,|s|)}\}$
and $q =$
```
match_(s,j,|s|) : σ̂ (y_i,y_{i+1},y_{i+2})
  = if y_{i+1}=y_{i+2}
    then -1
    else compare_(s,j,|s|) (y_i,y_{i+1},y_{i+2})
```

**case** $f = \texttt{match}_{(s,|s|,|s|)}$: Then

$$(\downarrow_{\widehat{\sigma}'} ([\![\mathit{STAGED}]\!]_{\mathsf{R}}(\texttt{match}))(\texttt{match}_{(s,|s|,|s|)}, (s, |s|, |s|)))\,\delta = \lfloor (q, \delta') \rfloor$$

where $\delta'(\texttt{seen}) = Z$ and $q = \texttt{match}_{(s,|s|,|s|)}\ \texttt{: } \underline{\widehat{\sigma}}\ \texttt{(}y_i\texttt{,}y_{i+1}\texttt{,}y_{i+2}\texttt{) = k-}\downarrow_{\texttt{int}}(|s|)$

**case** $f = \texttt{compare}_{(s,0,|s|)}$: Then, noting that $|s| \neq 0$,

$$(\downarrow_{\widehat{\sigma}} ([\![\mathit{STAGED}]\!]_{\mathsf{R}}(\texttt{compare}))(\texttt{compare}_{(s,0,|s|)}, (s, 0, |s|)))\,\delta = \lfloor (q, \delta') \rfloor$$

where $\delta'(\texttt{seen}) = Z \cup \{\texttt{match}_{(s,0,|s|)}, \texttt{match}_{(s,1,|s|)}\}$
and $q =$
```
compare_(s,0,|s|) : σ̂ (y_i,y_{i+1},y_{i+2})
  = if ↓char(ref(s,0))=ref(y_i,y_{i+1})
    then match_(s,1,|s|) (y_i,y_{i+1}+1,y_{i+2})
    else match_(s,0,|s|) (y_i,y_{i+1}+1,y_{i+2})
```

**case** $f = \texttt{compare}_{(s,j,|s|)}, j \in [1..|s|-1]$: This case is slightly more complicated, since we first need to know how the symbolic unfolding of `rematch` behaves (essentially [1, Lemma 3]). Define the termination relation $Y$ as

$$Y = \{(kp - jp, kp) \mid 0 \leq jp \leq kp \leq j\}$$

ordered by $(a_1, b_1) >_Y (a_2, b_2) \Leftrightarrow a_1 < a_2 \vee (a_1 = a_2 \wedge b_1 < b_2)$. Note that $(j, j)$ is thus the least element. By well-founded induction on $Y$, it is now easy to show that

$$[\![\mathit{STAGED}]\!]_{\mathsf{R}}(\texttt{rematch})(s, y_i, j, y_{i+1}, jp, kp, lp, y_{i+2})\,\delta = \lfloor (e, \delta') \rfloor$$

for some $e$ and $\delta'$, where either $e = \texttt{match}_{(s,0,|s|)}\texttt{(}y_i\texttt{,}y_{i+1}\texttt{+1,}y_{i+2}\texttt{)}$ and $\delta'(\texttt{seen}) = Z \cup \{\texttt{match}_{(s,0,|s|)}\}$ or $e = \texttt{compare}_{(s,jp',|s|)}\texttt{(}y_i\texttt{,}y_{i+1}\texttt{,}y_{i+2}\texttt{)}$ and $\delta'(\texttt{seen}) = Z \cup \{\texttt{compare}_{(s,jp',|s|)}\}$, for some $jp' \in [0..j]$.

Hence,

$$(\downarrow_{\widehat{\sigma}} ([\![\mathit{STAGED}]\!]_{\mathsf{R}}(\texttt{compare}))(\texttt{compare}_{(s,j,|s|)}, (s, j, |s|)))\,\delta = \lfloor (q, \delta'') \rfloor$$

where $\delta''(\texttt{seen}) = \delta'(\texttt{seen}) \cup \{\texttt{match}_{(s,j+1,|s|)}\}$

$$\text{and } q = \quad \texttt{compare}_{(s,j,|s|)} \;:\; \underline{\widehat{\sigma}}\ (y_i, y_{i+1}, y_{i+2}) \\ \qquad = \texttt{if}\ \downarrow_{\texttt{char}}(ref(s,j)) = \texttt{ref}(y_i, y_{i+1}) \\ \qquad\quad \texttt{then}\ \texttt{match}_{(s,j+1,|s|)}(y_i, y_{i+1}+\texttt{1}, y_{i+2}) \\ \qquad\quad \texttt{else}\ e$$

Now, by Lemma 4 and the definition of $[\![\mathit{STAGED}]\!]_{\mathsf{R}}$, there is some $p$ such that $[\![\mathit{STAGED}]\!]_{\mathsf{R}}\, s = \lfloor p \rfloor$. By Theorem 1, parts (1.) and (2.) follow.

Let $p = \texttt{local}\, d\, \texttt{in}\, f'$. By Lemma 3(1), every $q \in d$ is of the form

$$(\downarrow_{\widehat{\sigma}}([\![\mathit{STAGED}]\!]_{\mathsf{R}}(f))(f_{(v_1,\ldots,v_k)}, (v_1, \ldots, v_k)))\, \delta = \lfloor (q, \delta') \rfloor$$

where $f_{(v_1,\ldots,v_k)} \in X' \subseteq X$ and $\delta(\texttt{seen}) \subseteq X'$. Hence, we can observe by the above case analysis that the size of each $q$ is independent of $s$. Since $X$ contains at most $2|s| + 2$ such equations, we have established part (3.), namely $|p| \in O(|s|)$. □

Since a formal notion of "time" is outside the scope of our denotational framework, we will have to settle for some informal remarks concerning the efficiency of the specialized programs.

Given any $s$, it is a simple observation from the proof of Theorem 2 that when running the specialized program only a constant (i.e., independent of $|s|$) number of elementary operations are performed between each comparison, and that these comparisons are preserved exactly from the binding-time-improved program. From previous work [1], we know that this sequence of comparisons performed is identical to the sequence performed by the real Knuth-Morris-Pratt algorithm. Hence, the specialized program is at least as fast (asymptotically) as the linear-time Knuth-Morris-Pratt algorithm.

Note however that we use a different formal semantics than in previous work [1]. Therefore, even if we had a formal notion of time, we could not use the previous result directly.

# 4 Related work

Our closest related work is Filinski's formalizations of type-directed (monovariant) partial evaluation [14, 15, 19, 21, 22]. In fact, the technical aspects of the present work are inspired by Filinski's work: for call-by-value type-directed partial evaluation with computational effects for the simply-typed lambda calculus with constants, he uses a denotational semantics parameterized by a monad. The semantic construction of type-directed partial evaluation is a prime application of normalization by evaluation [6, 7].

Not only do the correctness proofs involve more advanced techniques, e.g., logical relations due to being in a higher-order setting, but the elegant semantic construction underlying type-directed partial evaluation is on closer inspection quite different from ours: the reification functions are defined by induction on their types, and the residualizing semantics is an instance of a parameterizable, conventional monadic semantics. The latter property is the key to its efficiency, since it can therefore delegate evaluation to existing, optimizing compilers; here, the types are a crucial part of the construction (cf. the untyped setting [23]). Type-directed partial evaluation also guarantees *completeness*, namely that if a solution (i.e., normal form) exists at all then the algorithm will also find one (cf. our Lemma 4). Conceivably, one could mimic this construction by defining (syntactic) polyvariant normal forms with a polyvariant reduction system, and prove the polyvariant specializer complete in the above sense.

Despite the differences, type-directed partial evaluation does however not seem incompatible with polyvariance. It seems possible to introduce a new dynamic "polyvariant" fixed-point constant to the current framework, and instantiating the monadic semantics with a more complicated monad capable of handling the necessary bookkeeping, similarly to our $T$. The interpretation of the new constant would then compute the transitive closure, possibly restricting polyvariance to a first-order fragment only using typing (in contrast to using the syntax grammar, as in our work).

Hatcliff and Danvy [30] have formalized a (monovariant) partial evaluator for Moggi's computational metalanguage. The partial evaluator is proven sound with repect to an underlying

operational semantics. Their formalization is evaluation-order independent and captures so-called control-based binding-time improvements (e.g., let-rearrangements) via the monadic laws.

In a small-step operational setting, they make the technical shortcut that "stuck" terms, i.e., irreducible non-values, are disallowed (and thus avoid two distinct notions of partiality, arising from both "stuck" terms and infinitely-reducing terms). This choice implies, in particular, that function constants must denote pure total functions. For example, using natural numbers, they consequently *define* $pred(0) = 0$, i.e., $0 - 1 = 0$, to avoid $pred(0)$ being "stuck". In contrast, we explitly allow both static and dynamic primitive functions to be partial.

Later, Lawall and Thiemann re-investigated sound (monovariant) specialization, also from an operational standpoint [36]. Finally, Ager, Danvy, and the author's treatment of the Knuth-Morris-Pratt algorithm is based on a formal semantic foundation [1]; this work was mentioned in more detail in Section 3.

# 5 Perspectives

The present work suggests several research directions.

## 5.1 Computational effects

A natural generalization is to allow arbitrary computational effects by parameterizing the source language by a monad. We not only gain expressibility, but can also keep track of dynamic computations more accurately. For the Knuth-Morris-Pratt algorithm, by the sheer possibility of making (dynamic) references to the text effectful, the residualizing semantics is forced by soundness to preserve them. Hence, the preservation of the algorithm's trace (the sequence of these references) will be guaranteed by the specializer [1].

Other natural generalizations would be to allow higher-order functions or partially-static data structures [33]. A promising direction would be to try to integrate polyvariance into type-directed partial evaluation, as suggested in Section 4.

## 5.2 Selective bounded static variation

Another natural generalization is to parametrize the residualizing semantics by a monad (layered on top of the source monad). This would allow more radical program rearrangements, e.g., via the continuation monad [19]. In particular, an "unlift" operation, @, (namely bounded static variation) can be defined:

$$\frac{\widehat{\Psi}, \widehat{\Gamma} \vdash \widehat{e} : \underline{\tau} \quad |[\![\|\underline{\tau}\|]\!]| < \infty}{\widehat{\Psi}, \widehat{\Gamma} \vdash @\widehat{e} : \overline{\tau}}$$

This rule will give rise to a lookup table in the residual program. In the realm of string matching, Danvy and the author have discovered that the key *bad-character-shift* heuristic of the efficient Boyer-Moore algorithm [10, 31] can be seen as an instance of bounded static variation [17].

## 5.3 Resource-bounded partial evaluation

If we are given only limited (insufficient) resources to compute the residual program, we may consider stopping the partial evaluator prematurely. The result will then be the already completed specialized functions, some trivially specialized functions (i.e., the pending functions and values, which are scheduled for specialization), and the original program [16]. Note that such an algorithm is indifferent to whether or not the number of specialized functions is finite. We expect that the current proof can be adapted without much difficulty.

The real challenge will be to decide what should be specialized [16, 18]. Naively, we can view the pending list as a priority queue, taking the priority to be simply be the number of potential callers. Advanced profiling methods are probably preferable, particularly if they take selective bounded static variation into account. Given a sufficiently powerful profiling analysis, the idea in

a nutshell is thus to focus the specialization effort (enhanced by tabulation) where it is expected to pay off most.

# 6 Conclusion

We have presented the first formal correctness proof of an offline polyvariant specialization algorithm for first-order recursive equations. Our approach has been to formulate the algorithm as a non-standard denotational semantics for the first-order language, and we have been able to establish correctness using mainly structural induction and fixed-point induction. The main technical challenges have been to keep track of the transitive closure and to tame symbolic unfolding.

As an application, the specialization algorithm has been shown to generate a program implementing the search phase of the Knuth-Morris-Pratt algorithm from an inefficient, but binding-time-improved, string matcher. Here, our formalization sharpens the existing analysis of information propagation.

The present work suggests several extensions, as already mentioned in Section 5. It is also a step towards a formalization of more advanced program transformations, such as Supercompilation and Generalized Partial Computation.

**Acknowledgments** A special thanks to Andrzej Filinski for introducing me to formal semantics; for encouragement; and for countless deep constructive comments. I am also grateful to Olivier Danvy for his insightful comments.

# References

[1] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. On Obtaining Knuth, Morris, and Pratt's String Matcher by Partial Evaluation. In Wei-Ngan Chin, editor, *Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 32–46. ACM Press, September 2002. Extended version available as technical report BRICS-RS-02-32.

[2] Mads Sig Ager, Olivier Danvy, and Henning Korsholm Rohde. Fast Partial Evaluation of Pattern Matching in Strings. *ACM Transactions on Programming Languages and Systems*, 2005. Available as technical report BRICS-RS-04-40. To appear.

[3] Torben Amtoft. *Sharing of Computations*. PhD thesis, University of Aarhus, Denmark, 1993. Technical report PB-453.

[4] Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær. The abstraction and instantiation of string-matching programs. In Mogensen et al. [37], pages 332–357.

[5] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 4–12, San Antonio, Texas, January 1999. Available online at `http://www.brics.dk/~pepm99/programme.html`.

[6] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *Lecture Notes in Computer Science*, pages 117–137. Springer-Verlag, 1998.

[7] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed $\lambda$-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[8] Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. North-Holland, 1988.

[9] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[10] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[11] Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[12] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, January 1989.

[13] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.

[14] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[15] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

[16] Olivier Danvy, Nevin C. Heintze, and Karoline Malmkjær. Resource-bounded partial evaluation. *ACM Computing Surveys*, 28(2):329–332, June 1996.

[17] Olivier Danvy and Henning Korsholm Rohde. On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation. *Information Processing Letters*. To appear.

[18] Saumya Debray. Resource-bounded partial evaluation. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 179–192, Amsterdam, The Netherlands, June 1997. ACM Press.

[19] Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

[20] Andrzej Filinski. Semantics of functional programming. Lecture notes, December 1998.

[21] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag.

[22] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[23] Andrzej Filinski and Henning Korsholm Rohde. Denotational Aspects of Untyped Normalization by Evaluation. *RAIRO - Theoretical Informatics and Applications*, 39(3):423–454, July 2005. FOSSACS'04 Special Issue. Extended version available as technical report BRICS-RS-05-4.

[24] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[25] Yoshihiko Futamura, Zenjiro Konishi, and Robert Glück. WSDFU: Program transformation system based on generalized partial computation. In Mogensen et al. [37], pages 358–378.

[26] Yoshihiko Futamura and Kenroku Nogi. Generalized partial computation. In Bjørner et al. [8], pages 133–151.

[27] Yoshihiko Futamura, Kenroku Nogi, and Akihiko Takano. Essence of generalized partial computation. *Theoretical Computer Science*, 90(1):61–79, 1991.

[28] Robert Glück and Andrei Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy, editors, *Proceedings of the Third International Workshop on Static Analysis WSA'93*, number 724 in Lecture Notes in Computer Science, pages 112–123, Padova, Italy, September 1993. Springer-Verlag.

[29] Bernd Grobauer and Julia L. Lawall. Partial evaluation of pattern matching in strings, revisited. *Nordic Journal of Computing*, 8(4):437–462, 2002.

[30] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, pages 507–541, 1997.

[31] R. Nigel Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10(6):501–506, 1980.

[32] Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In Bjørner et al. [8], pages 225–282.

[33] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, London, UK, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/`.

[34] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[35] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[36] Julia Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In Martin Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, 3rd International Symposium, TACS'97*, number 1281 in Lecture Notes in Computer Science, pages 227–238, Sendai, Japan, September 1997. Springer-Verlag.

[37] Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors. *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[38] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[39] Henning Korsholm Rohde. Measuring the propagation of information in partial evaluation. Technical Report BRICS-RS-05-26, BRICS, University of Aarhus, Denmark, August 2005.

[40] Jens Peter Secher. *Driving-based Program Transformation in Theory and Practice*. PhD thesis, Department of Computer Science, Copenhagen University, July 2002.

[41] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[42] Valentin F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

[43] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

# Recent BRICS Report Series Publications

**RS-05-34** **Henning Korsholm Rohde.** ***Formal Aspects of Polyvariant Specialization*****. November 2005. 27 pp.**

**RS-05-33** **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Sumit Nain.** ***Bisimilarity is not Finitely Based over BPA with Interrupt.*** **October 2005. 33 pp. This paper supersedes BRICS Report RS-04-24. An extended abstract of this paper appeared in** ***Algebra and Coalgebra in Computer Science, 1st Conference, CALCO 2005*****, Swansea, Wales, 3–6 September 2005, Lecture Notes in Computer Science 3629, pp. 54–68, Springer-Verlag, 2005.**

**RS-05-32** **Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach.** ***Static Validation of XSL Transformations*****. October 2005. 50 pp.**

**RS-05-31** **Christian Kirkegaard and Anders Møller.** ***Type Checking with XML Schema in*** **XACT. September 2005. 20 pp.**

**RS-05-30** **Karl Krukow.** ***An Operational Semantics for Trust Policies*****. September 2005. 38 pp.**

**RS-05-29** **Olivier Danvy and Henning Korsholm Rohde.** ***On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation*****. September 2005. ii+9 pp. To appear in** ***Information Processing Letters*****. This version supersedes BRICS RS-05-14.**

**RS-05-28** **Jiří Srba.** ***On Counting the Number of Consistent Genotype Assignments for Pedigrees*****. September 2005. 15 pp. To appear in FSTTCS '05.**

**RS-05-27** **Pascal Zimmer.** ***A Calculus for Context-Awareness*****. August 2005. 21 pp.**

**RS-05-26** **Henning Korsholm Rohde.** ***Measuring the Propagation of Information in Partial Evaluation*****. August 2005. 39 pp.**

**RS-05-25** **Dariusz Biernacki and Olivier Danvy.** ***A Simple Proof of a Folklore Theorem about Delimited Control*****. August 2005. ii+11 pp. To appear in** ***Journal of Functional Programming*****. This version supersedes BRICS RS-05-10.**