A Semantic Account of Type-Directed Partial Evaluation

Principles and Practice of Declarative Programming 1999, LNCS 1702

Andrzej Filinski[⋆]

BRICS[⋆⋆], Dept. of Computer Science
University of Aarhus, Denmark
`andrzej@brics.dk`

Abstract. We formally characterize partial evaluation of functional programs as a normalization problem in an equational theory, and derive a type-based normalization-by-evaluation algorithm for computing normal forms in this setting. We then establish the correctness of this algorithm using a semantic argument based on Kripke logical relations. For simplicity, the results are stated for a non-strict, purely functional language; but the methods are directly applicable to stating and proving correctness of type-directed partial evaluation in ML-like languages as well.

1 Introduction

The goal of partial evaluation (PE) is as follows: given a program $\vdash p : S \times D \to R$ of two arguments, and a fixed "static" argument $s : S$, produce a *specialized* program $\vdash p_s : D \to R$ such that for all "dynamic" $d : D$, $\mathit{Eval}\big(p_s\, d\big) = \mathit{Eval}\big(p(s,d)\big)$. That is, running the specialized program on the dynamic argument is equivalent to running the original program on both the static and the dynamic one.

In a functional language, it is of course trivial to come up with such a p_s: just take $p_s = \lambda d. p(s,d)$. That is, the specialized program simply invokes the original program with a constant first argument. But such a p_s is likely to be suboptimal: the knowledge of s may already allow us to perform some simplifications that are independent of d. For example, consider the power function:

$$\mathit{power}\,(n,x) \overset{\text{rec}}{=} \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } x \times \mathit{power}(n-1,x)$$

Suppose we want to compute the third power of several numbers. We can achieve this using the trivially specialized program:

$$\mathit{power}_3 = \lambda x. \mathit{power}(3,x)$$

But using a few simple rules derived from the semantics of the language, we can safely transform power_3 to the much more efficient

$$\mathit{power}'_3 = \lambda x. x \times (x \times (x \times 1))$$

[⋆] Part of this work was carried out at the Laboratory for Foundations of Computer Science, University of Edinburgh, supported by a EuroFOCS research fellowship.

[⋆⋆] Basic Research in Computer Science (`www.brics.dk`),
Centre of the Danish National Research Foundation

Using further arithmetic identities, we can also easily eliminate the multiplication by 1. On the other hand, if only the argument x were known, we could not simplify much: the specialized program would in general still need to contain a recursive definition and a conditional test – in addition to the multiplication. (Note that, even when x is 0 or 1, the function as defined should still diverge for negative values of n.)

To facilitate automation of the task, partial evaluation is often expressed as a two-phase process, usually referred to as *off-line* PE [14]:

1. A *binding-time annotation* phase, which identifies all the operations that can be performed using just the static input. This can be done either mechanically by a *binding-time analysis* (often based on abstract interpretation), or – if the intended usage of the program is clear and the annotations are sufficiently intuitive and non-intrusive – as part of the original program.
2. A *specialization* phase, which takes the annotated program and the static input, and produces a simplified p_s, in which all the operations marked as static have been eliminated.

The annotations must of course be consistent, i.e., a subcomputation in the program cannot be classified as static if its result can not necessarily be found from only the static input. But they may be conservative by classifying some computations as dynamic even if they could in fact be performed at specialization time. Techniques for accurate binding-time analysis have been studied extensively [14]. In the following we will therefore limit our attention to the second phase, i.e., to efficiently specializing programs that are already binding-time separated.

A particularly simple way of phrasing specialization is as a general-purpose simplification of the trivially specialized program $\lambda d.\, p\,(s, d)$: contracting β-redexes and eliminating static operations as their inputs become known. What makes this approach attractive is the technique of "reduction-free normalization" or "normalization by evaluation", already known from logic and category theory [2, 3, 7]. A few challenges arise, however, with extending these results to a programming-language setting. Most notably:

- *Interpreted* base types and their associated static operations. These need to be properly accounted for, in addition to the β-reduction.
- Unrestricted recursion. This prevents a direct application of the usual strong-normalization results. That is, not every well-typed term even has a normal form; and not every reduction strategy will find it when it does exist.
- Call-by-value languages, and effects other than non-termination. In such a setting, the usual $\beta\eta$-conversions are actually unsound: unrestricted rearrangement of side effects may completely change the meaning of a program.

We will treat the first two concerns in detail. The call-by-value case uses the same principles, but for space reasons we will only briefly outline the necessary changes.

The paper is organized as follows: Section 2 introduces our programming language, the notion of a binding-time separated signature, and our desiderata for a partial evaluator; Section 3 presents the type-directed partial evaluation algorithm; and Section 4 shows its correctness with respect to the criteria in Section 2.

Finally, Section 5 presents a few variations and extensions, and Section 6 concludes and outlines directions for further research.

2 A Small Language

2.1 The framework and one-level language

Our prototypical functional language has the following syntax of types and terms:

$$\begin{aligned}\sigma &::= b \mid \sigma_1 \rightarrow \sigma_2 \\ E &::= l \mid c_{\sigma_1,\ldots,\sigma_n} \mid x \mid \lambda x^\sigma . E \mid E_1\, E_2\end{aligned}$$

Here b ranges over a set of base types listed in some signature Σ, l over a set $\Xi(b)$ of literals (numerals, truth values, etc.) for each base type b, and c over a set of (possibly polymorphic) function constants in Σ. Adding finite-product types would be completely straightforward throughout the paper, but we omit this extension for conciseness.

A typing context Γ is a finite mapping of variable names to well-formed types over Σ. The typing rules for terms are then standard:

$$\frac{l \in \Xi(b)}{\Gamma \vdash_\Sigma l : b} \qquad \frac{\Sigma(c_{\sigma_1,\ldots,\sigma_n}) = \sigma}{\Gamma \vdash_\Sigma c_{\sigma_1,\ldots,\sigma_n} : \sigma} \qquad \frac{\Gamma(x) = \sigma}{\Gamma \vdash_\Sigma x : \sigma}$$

$$\frac{\Gamma, x{:}\,\sigma_1 \vdash_\Sigma E : \sigma_2}{\Gamma \vdash_\Sigma \lambda x^{\sigma_1} . E : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Gamma \vdash_\Sigma E_1 : \sigma_1 \rightarrow \sigma_2 \qquad \Gamma \vdash_\Sigma E_2 : \sigma_1}{\Gamma \vdash_\Sigma E_1\, E_2 : \sigma_2}$$

An *interpretation* of a signature Σ is a triple $\mathcal{I} = (\mathcal{B}, \mathcal{L}, \mathcal{C})$. $\mathcal{B}$ maps every base type b in Σ to a predomain (i.e., a bottomless cpo, usually discretely ordered). Then we can interpret every type phrase σ over Σ as a domain (pointed cpo):

$$\begin{aligned}[\![b]\!]^{\mathcal{B}} &= \mathcal{B}(b)_\bot \\ [\![\sigma_1 \rightarrow \sigma_2]\!]^{\mathcal{B}} &= [\![\sigma_1]\!]^{\mathcal{B}} \rightarrow [\![\sigma_2]\!]^{\mathcal{B}}\end{aligned}$$

where the interpretation of an arrow type is the full continuous function space. We also define the meaning of a typing assignment Γ as a labelled product of the domains interpreting the types of individual variables,

$$[\![\Gamma]\!]^{\mathcal{B}} = \prod\nolimits_{x \in \mathrm{dom}\,\Gamma} [\![\Gamma(x)]\!]^{\mathcal{B}} .$$

Further, for any base type b and literal $l \in \Xi(b)$, the interpretation must specify an element $\mathcal{L}_b(l) \in \mathcal{B}(b)$; and for every type instance of a polymorphic constant, an element $\mathcal{C}(c_{\sigma_1,\ldots,\sigma_n}) \in [\![\Sigma(c_{\sigma_1,\ldots,\sigma_n})]\!]^{\mathcal{B}}$. Then we interpret a well-typed term $\Gamma \vdash_\Sigma E : \sigma$ as a (total) continuous function $[\![E]\!]^{\mathcal{I}} : [\![\Gamma]\!]^{\mathcal{B}} \rightarrow [\![\sigma]\!]^{\mathcal{B}}$,

$$\begin{aligned}[\![l]\!]^{\mathcal{I}} \rho &= \mathbf{val}^\bot\, \mathcal{L}_b(l) \\ [\![c_{\sigma_1,\ldots,\sigma_n}]\!]^{\mathcal{I}} \rho &= \mathcal{C}(c_{\sigma_1,\ldots,\sigma_n}) \\ [\![x]\!]^{\mathcal{I}} \rho &= \rho\, x \\ [\![\lambda x^\sigma . E]\!]^{\mathcal{I}} \rho &= \lambda a^{[\![\sigma]\!]^{\mathcal{B}}} . [\![E]\!]^{\mathcal{I}} (\rho[x \mapsto a]) \\ [\![E_1\, E_2]\!]^{\mathcal{I}} \rho &= [\![E_1]\!]^{\mathcal{I}} \rho\, ([\![E_2]\!]^{\mathcal{I}} \rho)\end{aligned}$$

(We use the following notation: $\mathbf{val}^{\perp}\, x$ and $\mathbf{let}^{\perp}\; y \Leftarrow x \textbf{ in } f\,y$ are lifting-injection and the strict extension of f, respectively; $b \to x \,[\!]\, y$ chooses between x and y based on the truth value b.)

When $\vdash_{\Sigma} E : b$ is a closed term of base type, we define the partial function $\mathit{Eval}_{\mathcal{I}}$ by $\mathit{Eval}_{\mathcal{I}}(E) = n$ if $[\![E]\!]^{\mathcal{I}}\emptyset = \mathbf{val}^{\perp}\, n$ and undefined otherwise.

Definition 1 (standard static language). *We define a simple functional language (essentially PCF [17]) by taking the signature* Σ_{s} *as follows. The base types are* int *and* bool*; the literals,* $\Xi(\mathsf{int}) = \{\ldots, \text{-}\mathsf{1}, \mathsf{0}, \mathsf{1}, \mathsf{2}, \ldots\}$ *and* $\Xi(\mathsf{bool}) = \{\mathsf{true}, \mathsf{false}\}$*; and the constants,*

$$\begin{array}{rlrl} +, -, \times &: \mathsf{int} \to \mathsf{int} \to \mathsf{int} & \qquad \mathsf{if}_{\sigma} &: \mathsf{bool} \to \sigma \to \sigma \to \sigma \\ =, < &: \mathsf{int} \to \mathsf{int} \to \mathsf{bool} & \mathsf{fix}_{\sigma} &: (\sigma \to \sigma) \to \sigma \end{array}$$

(We write the binary operations infixed for readability.) The interpretation of this signature is also as expected:

$$\begin{aligned} \mathcal{B}_{\mathrm{s}}(\mathsf{bool}) &= \mathbf{B} = \{\mathit{tt}, \mathit{ff}\} \\ \mathcal{B}_{\mathrm{s}}(\mathsf{int}) &= \mathbf{Z} = \{\ldots, -1, 0, 1, 2, \ldots\} \end{aligned}$$

$$\begin{aligned} \mathcal{C}_{\mathrm{s}}(\star) &= \lambda x^{\mathbf{Z}_{\perp}}.\lambda y^{\mathbf{Z}_{\perp}}.\mathbf{let}^{\perp}\; n \Leftarrow x \textbf{ in } \mathbf{let}^{\perp}\; m \Leftarrow y \textbf{ in } \mathbf{val}^{\perp}\, m \star n \quad {\scriptstyle \star \in \{+,-,\times,=,<\}} \\ \mathcal{C}_{\mathrm{s}}(\mathsf{if}_{\sigma}) &= \lambda x^{\mathbf{B}_{\perp}}.\lambda a_1^{[\![\sigma]\!]}.\lambda a_2^{[\![\sigma]\!]}.\mathbf{let}^{\perp}\; b \Leftarrow x \textbf{ in } b \to a_1 \,[\!]\, a_2 \\ \mathcal{C}_{\mathrm{s}}(\mathsf{fix}_{\sigma}) &= \lambda f^{[\![\sigma]\!] \to [\![\sigma]\!]}.\bigsqcup\nolimits_{i \in \omega} f^i \perp_{[\![\sigma]\!]} \end{aligned}$$

It is well known (computational adequacy of the denotational semantics for call-by-name evaluation [17]) that with this interpretation, $\mathit{Eval}_{\mathcal{I}_{\mathrm{s}}}$ is computable.

2.2 The binding-time separated language

Assume now that the signature Σ is partitioned according to binding times, $\Sigma = \Sigma_{\mathrm{s}}, \Sigma_{\mathrm{d}}$. We will write type and term constants from the static part overlined, and the dynamic ones underlined. For simplicity, we require that the dynamic base types do not come with any new literals, i.e., $\Xi(\underline{b}) = \emptyset$. (If needed, they can be added as dynamic constants.) However, some base types will be persistent, i.e., have both static and dynamic versions with the same intended meaning. In that case, we also include *lifting functions* $\$_b : \overline{b} \to \underline{b}$ in the dynamic signature.

We say that a type τ is *fully dynamic* if it is constructed from dynamic base types only,

$$\tau ::= \underline{b} \mid \tau_1 \to \tau_2$$

We also reserve Δ for typing assumptions assigning fully dynamic types to all variables. All term constants in Σ_{d} must have fully dynamic types, and in particular, polymorphic dynamic constants must only be instantiated by dynamic types, e.g., $\Sigma_{\mathrm{d}}(\underline{\mathsf{if}}_{\tau}) = \underline{\mathsf{bool}} \to \tau \to \tau \to \tau$.

We will always take the language from Definition 1 with the standard semantics $\mathcal{I}_{\mathrm{s}}$ as the static part. The dynamic signature typically also has some intended *evaluating interpretation* $\mathcal{I}_{\mathrm{d}}^{\mathrm{e}}$; in particular, when Σ_{d} is merely a copy of Σ_{s}, we can use $\mathcal{I}_{\mathrm{s}}$ directly for $\mathcal{I}_{\mathrm{d}}^{\mathrm{e}}$ (interpreting all lifting functions as identities). Later, however, we will also introduce a "code-generating", *residualizing* interpretation.

Example 1. Here are the four different binding-time annotations for the function *power* : $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ (abbreviating int as ι):

$$
\begin{aligned}
power_{ss} &: \overline{\iota} \to \overline{\iota} \to \overline{\iota} = \lambda x^{\overline{\iota}}.\, \overline{\mathsf{fix}}_{\overline{\iota} \to \overline{\iota}}\, (\lambda p^{\overline{\iota} \to \overline{\iota}}.\, \lambda n^{\overline{\iota}}.\, \overline{\mathsf{if}}_{\overline{\iota}}\, (n \mathbin{\overline{=}} 0)\, 1\, (x \mathbin{\overline{\times}} p\, (n \mathbin{\overline{-}} 1)))\\
power_{sd} &: \overline{\iota} \to \underline{\iota} \to \underline{\iota} = \lambda x^{\overline{\iota}}.\, \underline{\mathsf{fix}}_{\underline{\iota} \to \underline{\iota}}\, (\lambda p^{\underline{\iota} \to \underline{\iota}}.\, \lambda n^{\underline{\iota}}.\, \underline{\mathsf{if}}_{\underline{\iota}}\, (n \mathbin{\underline{=}} \$\, 0)\, (\$\, 1)\, (\$\, x \mathbin{\underline{\times}} p\, (n \mathbin{\underline{-}} \$\, 1)))\\
power_{ds} &: \underline{\iota} \to \overline{\iota} \to \underline{\iota} = \lambda x^{\underline{\iota}}.\, \overline{\mathsf{fix}}_{\overline{\iota} \to \underline{\iota}}\, (\lambda p^{\overline{\iota} \to \underline{\iota}}.\, \lambda n^{\overline{\iota}}.\, \overline{\mathsf{if}}_{\underline{\iota}}\, (n \mathbin{\overline{=}} 0)\, (\$\, 1)\, (x \mathbin{\underline{\times}} p\, (n \mathbin{\overline{-}} 1)))\\
power_{dd} &: \underline{\iota} \to \underline{\iota} \to \underline{\iota} = \lambda x^{\underline{\iota}}.\, \underline{\mathsf{fix}}_{\underline{\iota} \to \underline{\iota}}\, (\lambda p^{\underline{\iota} \to \underline{\iota}}.\, \lambda n^{\underline{\iota}}.\, \underline{\mathsf{if}}_{\underline{\iota}}\, (n \mathbin{\underline{=}} \$\, 0)\, (\$\, 1)\, (x \mathbin{\underline{\times}} p\, (n \mathbin{\underline{-}} \$\, 1)))
\end{aligned}
$$

Note how the fixed-point and conditional operators are classified as static or dynamic, depending on the binding time of the second argument.

2.3 Static normal forms and PE

Definition 2 (static normal forms). *Among the well-typed, purely dynamic terms* $\Delta \vdash_{\Sigma_{\mathrm{d}}} E : \tau$, *we distinguish those in* normal *and* atomic *form:*

$$
\frac{\Delta \vdash^{\mathrm{at}} E : \underline{b}}{\Delta \vdash^{\mathrm{nf}} E : \underline{b}}
\qquad
\frac{\Delta, x{:}\,\tau_1 \vdash^{\mathrm{nf}} E : \tau_2}{\Delta \vdash^{\mathrm{nf}} \lambda x^{\tau_1}.\, E : \tau_1 \to \tau_2}\, {\scriptstyle x \notin \mathrm{dom}\, \Delta}
$$

$$
\frac{l \in \Xi(\overline{b})}{\Delta \vdash^{\mathrm{at}} \$_b\, l : \underline{b}}
\qquad
\frac{\Sigma_{\mathrm{d}}(\underline{c}_{\tau_1,\ldots,\tau_n}) = \tau}{\Delta \vdash^{\mathrm{at}} \underline{c}_{\tau_1,\ldots,\tau_n} : \tau}
\qquad
\frac{\Delta(x) = \tau}{\Delta \vdash^{\mathrm{at}} x : \tau}
$$

$$
\frac{\Delta \vdash^{\mathrm{at}} E_1 : \tau_1 \to \tau_2 \qquad \Delta \vdash^{\mathrm{nf}} E_2 : \tau_1}{\Delta \vdash^{\mathrm{at}} E_1\, E_2 : \tau_2}
$$

In particular, such terms contain no static constants nor β-redexes. (Incidentally, this also means that if we had included polymorphic lets in the source language, they would simply get unfolded in the resulting normal forms.)

We can now define a notion of normalization based on (undirected) equality, rather than on (directed) reduction [3]. Since lambda-abstracting a dynamic-type term over a dynamic-type variable still yields a dynamic term, it suffices to be able to compute normal forms of closed terms:

Definition 3 (static equivalence and normalization). *Let* $\mathcal{I}_{\mathrm{s}}$ *be an interpretation of* Σ_{s}. *We say that two terms* $\vdash_{\Sigma_{\mathrm{s}},\Sigma_{\mathrm{d}}} E : \sigma$ *and* $\vdash_{\Sigma_{\mathrm{s}},\Sigma_{\mathrm{d}}} E' : \sigma$ *are* statically equivalent wrt. $\mathcal{I}_{\mathrm{s}}$, *written* $E =^{\mathcal{I}_{\mathrm{s}}} E'$, *if for all* $\mathcal{I}_{\mathrm{d}}$ *interpreting* Σ_{d}, $[\![E]\!]^{\mathcal{I}_{\mathrm{s}},\mathcal{I}_{\mathrm{d}}} = [\![E']\!]^{\mathcal{I}_{\mathrm{s}},\mathcal{I}_{\mathrm{d}}}$. *A* static-normalization function *is then a computable partial function NF on well-typed terms such that*

1. *If* $\vdash_{\Sigma_{\mathrm{s}},\Sigma_{\mathrm{d}}} E : \tau$ *and* $NF(E) = \tilde{E}$ *then* $\vdash^{\mathrm{nf}}_{\Sigma_{\mathrm{d}}} \tilde{E} : \tau$ *and* $\tilde{E} =^{\mathcal{I}_{\mathrm{s}}} E$.
2. *If also* $\vdash_{\Sigma_{\mathrm{s}},\Sigma_{\mathrm{d}}} E' : \tau$ *and* $E' =^{\mathcal{I}_{\mathrm{s}}} E$ *then* $NF(E') \equiv NF(E)$ *(α-equivalence).*

We further say that such a normalization function is complete *if whenever an* $\tilde{E}$ *satisfying the conditions in (1) exists,* $NF(E)$ *is defined.*

Example 2. One can check that a complete static-normalization function *NF* for our language must have the following properties:

$$\begin{aligned} NF(\$\,(power_{ss}\,\mathsf{3}\,\mathsf{4})) &\equiv \$\,\mathsf{81} \\ NF(\lambda x^{\underline{\mathsf{int}}}.\,power_{ds}\,x\,\mathsf{3}) &\equiv \lambda x^{\underline{\mathsf{int}}}.\,x \mathbin{\underline{\times}} (x \mathbin{\underline{\times}} (x \mathbin{\underline{\times}} \$\,\mathsf{1})) \\ NF(\lambda x^{\underline{\mathsf{int}}}.\,power_{ds}\,x\,\mathsf{\text{-}2}) &\quad \text{undefined} \end{aligned}$$

Note first that ordinary evaluation is just a special case of static normalization. The second example shows how static normalization achieves the partial-evaluation goal of the introduction. Finally, some terms have no static normal form at all; in that case, the normalization function must diverge.

There are two basic ways to compute normal forms. The usual one is based on term rewriting, repeatedly locating and contracting β-redexes and applications of static constants (and possibly η-expanding the final result). But there is an alternative technique, *normalization by evaluation*, which utilizes the existing mechanism of complete-program evaluation (defined only for closed terms of base type) as the normalization engine for general terms. This is the subject of the next section.

3 A Normalization-by-Evaluation Algorithm

We now present Type-Directed Partial Evaluation (TDPE), an efficient algorithm for computing static normal forms.

3.1 Representing programs as data

To compute normal forms, we need a way of representing them as program outputs. Assume therefore that we have base cpos rich enough to contain unique representations of all well-formed dynamic types, variable names, and (open) static-normal form terms, i.e., sets T, $\mathcal{V}$, and Λ with injective operations

$$\begin{aligned} BASE_b &: T & CST &: \mathcal{V} \times T^* \to \Lambda \\ ARR &: T \times T \to T & VAR &: \mathcal{V} \to \Lambda \\ LIT_b &: \mathcal{B}_s(b) \to \Lambda & LAM &: \mathcal{V} \times T \times \Lambda \to \Lambda \\ & & APP &: \Lambda \times \Lambda \to \Lambda \end{aligned}$$

(where T^* is the set of finite lists of elements from T). Using these, we can define injective *representation functions* for types and terms, such that $\ulcorner\tau\urcorner \in T$ for any dynamic type τ, and $\ulcorner E\urcorner \in \Lambda$ for $\Delta \vdash^{\mathrm{nf}}_{\Sigma_\mathrm{d}} E : \tau$, by equations such as

$$\ulcorner\tau_1 \to \tau_2\urcorner = ARR(\ulcorner\tau_1\urcorner, \ulcorner\tau_2\urcorner) \qquad \ulcorner\lambda x^{\tau}.\,E\urcorner = LAM(x, \ulcorner\tau\urcorner, \ulcorner E\urcorner) \qquad \ulcorner\$_b\, l\urcorner = LIT_b(\mathcal{L}_b(l))$$

We do not need to require a priori that all elements of T and Λ represent well-formed types and terms (although this is easy to achieve), let alone well-typed ones, or ones in normal form. For example, we could simply take all of T, $\mathcal{V}$, and Λ as the type of finite character strings. Or, even more radically, Gödel-code everything in terms of integer arithmetic only.

To account for potentially diverging normalizations, we must now turn the set of term representations into a pointed cpo. To also model the generation of "new" variable names, however, we will not work with elements of $\Lambda_\perp$ directly, but instead introduce a term-family representation,

$$\hat{\Lambda} = \mathcal{N} \to \Lambda_\perp$$

where $\mathbf{N} \subseteq \mathcal{N}$. The intent is that for $e \in \hat{\Lambda}$ and $i \in \mathbf{N}$, if $e\,i = \mathbf{val}^\perp \ulcorner E \urcorner$ then all bound variables of E will belong to the set $\{\mathsf{g}_i, \mathsf{g}_{i+1}, \ldots\} \subseteq \mathcal{V}$.

We also define wrapper functions to conveniently build representations of lambda-terms without committing to particular choices of bound-variable names:

$$\begin{aligned}
\widehat{LIT}_b : \mathcal{B}_{\mathrm{s}}(b) \to \hat{\Lambda} &= \lambda n.\, \lambda i.\, \mathbf{val}^\perp\; LIT_b(n) \\
\widehat{CST} : \mathcal{V} \times T^* \to \hat{\Lambda} &= \lambda(c, \boldsymbol{t}).\, \lambda i.\, \mathbf{val}^\perp\; CST(c, \boldsymbol{t}) \\
\widehat{VAR} : \mathcal{V} \to \hat{\Lambda} &= \lambda v.\, \lambda i.\, \mathbf{val}^\perp\; VAR(v) \\
\widehat{LAM} : T \times (\mathcal{V} \to \hat{\Lambda}) \to \hat{\Lambda} &= \lambda(t, \varepsilon).\, \lambda i.\, \mathbf{let}^\perp\; l \Leftarrow \varepsilon\, \mathsf{g}_i\, (i+1)\; \mathbf{in}\; \mathbf{val}^\perp\; LAM(\mathsf{g}_i, t, l) \\
\widehat{APP} : \hat{\Lambda} \times \hat{\Lambda} \to \hat{\Lambda} &= \\
&\quad \lambda(e_1, e_2).\, \lambda i.\, \mathbf{let}^\perp\; l_1 \Leftarrow e_1\, i\; \mathbf{in}\; \mathbf{let}^\perp\; l_2 \Leftarrow e_2\, i\; \mathbf{in}\; \mathbf{val}^\perp\; APP(l_1, l_2)
\end{aligned}$$

(These definitions would not be needed in a setting with support for higher-order abstract syntax. But one of our goals is to show rigorously that all the variable-name manipulations can be done efficiently by the normalization algorithm itself, without relying on higher-level operations such as capture-avoiding substitution or higher-order matching.)

Example 3. Let $t = ARR(BASE_{\mathsf{int}}, BASE_{\mathsf{int}})$. Then

$$\begin{aligned}
&\widehat{LAM}\,(t, \lambda v^{\mathcal{V}}.\, \widehat{APP}\,(\widehat{VAR}\,v, \widehat{LIT}_{\mathsf{int}}\,3))\,7 \\
&\quad = \mathbf{val}^\perp\; LAM(\mathsf{g}_7, t, APP(VAR(\mathsf{g}_7), LIT_{\mathsf{int}}(3))) = \mathbf{val}^\perp \ulcorner \lambda \mathsf{g}_7^{\underline{\mathsf{int}} \to \underline{\mathsf{int}}}.\, \mathsf{g}_7\,(\$\,3) \urcorner
\end{aligned}$$

That is, we can apply an element of $\hat{\Lambda}$ constructed using the wrapper functions to a starting index and obtain the representation of a concrete lambda-term.

3.2 The residualizing interpretation

We now define a non-standard interpretation $\mathcal{I}_{\mathrm{d}}^{\mathrm{r}} = (\mathcal{B}_{\mathrm{d}}^{\mathrm{r}}, \emptyset, \mathcal{C}_{\mathrm{d}}^{\mathrm{r}})$ of the dynamic signature Σ_{d}, based on representations of syntactic program fragments and operations constructing such representations. We will abbreviate $[\![-]\!]^{\mathcal{I}_{\mathrm{s}}, \mathcal{I}_{\mathrm{d}}^{\mathrm{r}}}$ as $[\![-]\!]_{\mathrm{r}}$. For the interpretation of Σ_{d}'s types, we take

$$\mathcal{B}_{\mathrm{d}}^{\mathrm{r}}(\underline{b}) = \hat{\Lambda}\,.$$

This allows us to define for any dynamic τ a pair of continuous functions, often called *reification*, $\downarrow^\tau : [\![\tau]\!]_{\mathrm{r}} \to \hat{\Lambda}$, and *reflection*, $\uparrow_\tau : \hat{\Lambda} \to [\![\tau]\!]_{\mathrm{r}}$, as follows:

$$\begin{aligned}
\downarrow^{\underline{b}} &= \lambda t^{\hat{\Lambda}}.\, t & \downarrow^{\tau_1 \to \tau_2} &= \lambda f^{[\![\tau_1]\!]_{\mathrm{r}} \to [\![\tau_2]\!]_{\mathrm{r}}}.\, \widehat{LAM}\,(\ulcorner \tau_1 \urcorner, \lambda v^{\mathcal{V}}.\, \downarrow^{\tau_2} (f\,(\uparrow_{\tau_1} (\widehat{VAR}\,v)))) \\
\uparrow_{\underline{b}} &= \lambda e^{\hat{\Lambda}}.\, e & \uparrow_{\tau_1 \to \tau_2} &= \lambda e^{\hat{\Lambda}}.\, \lambda a^{[\![\tau_1]\!]_{\mathrm{r}}}.\, \uparrow_{\tau_2} (\widehat{APP}\,(e, \downarrow^{\tau_1} a))
\end{aligned}$$

Informally, reification constructs a syntactic representation of a "well-behaved" semantic value, while reflection constructs such values from pieces of syntax. For the residualizing interpretations of Σ_{d}'s term constants we now take

$$\mathcal{C}^{\mathrm{r}}_{\mathrm{d}}(\underline{c}_{\tau_1,\ldots,\tau_n}) = \uparrow_{\Sigma_{\mathrm{d}}(c_{\tau_1,\ldots,\tau_n})} (\widehat{CST}\,(c, [\ulcorner\tau_1\urcorner, \ldots, \ulcorner\tau_n\urcorner]))$$
$$\mathcal{C}^{\mathrm{r}}_{\mathrm{d}}(\$_b) = \lambda x^{\mathcal{B}_{\mathrm{s}}(b)_\perp}.\,\mathbf{let}^\perp\ n \Leftarrow x\ \mathbf{in}\ \widehat{LIT}_b\, n$$

That is, a general dynamic constant is simply interpreted as the reflection of its type-annotated name, while a lifting function forces evaluation of its argument and constructs a representation of the literal result. (It is this forcing of static subcomputations that may cause the whole specialization process to diverge.)

Example 4. Applying the reification function to the residualizing meaning of a term not in static normal form, we obtain:

$$\begin{aligned}
&\downarrow^{(\underline{\mathsf{int}}\to\underline{\mathsf{int}})\to\underline{\mathsf{int}}} (\llbracket(\lambda x^{\overline{\mathsf{int}}}.\,\lambda f^{\underline{\mathsf{int}}\to\underline{\mathsf{int}}}.\,f\,(\$_{\mathsf{int}}\,(x \mathbin{\overline{+}} 1)))\,2\rrbracket_{\mathrm{r}}\,\emptyset)\\
&\quad= \downarrow^{(\underline{\mathsf{int}}\to\underline{\mathsf{int}})\to\underline{\mathsf{int}}} (\llbracket\lambda f.\,f\,(\$_{\mathsf{int}}\,(x \mathbin{\overline{+}} 1))\rrbracket_{\mathrm{r}}\,(\emptyset[x \mapsto \llbracket 2\rrbracket_{\mathrm{r}}\,\emptyset]))\\
&\quad= \downarrow^{(\underline{\mathsf{int}}\to\underline{\mathsf{int}})\to\underline{\mathsf{int}}} (\lambda\varphi^{\hat{\Lambda}\to\hat{\Lambda}}.\,\llbracket f\,(\$_{\mathsf{int}}\,(x \mathbin{\overline{+}} 1))\rrbracket_{\mathrm{r}}\,(\emptyset[x \mapsto \mathbf{val}^\perp\,2, f \mapsto \varphi]))\\
&\quad= \downarrow^{(\underline{\mathsf{int}}\to\underline{\mathsf{int}})\to\underline{\mathsf{int}}} (\lambda\varphi.\,\varphi\,(\mathcal{C}^{\mathrm{r}}_{\mathrm{d}}(\$_{\mathsf{int}})\,(\mathcal{C}_{\mathrm{s}}(+)\,(\mathbf{val}^\perp\,2)\,(\mathbf{val}^\perp\,1))))\\
&\quad= \downarrow^{(\underline{\mathsf{int}}\to\underline{\mathsf{int}})\to\underline{\mathsf{int}}} (\lambda\varphi.\,\varphi\,(\mathcal{C}^{\mathrm{r}}_{\mathrm{d}}(\$_{\mathsf{int}})\,(\mathbf{val}^\perp\,3))) = \downarrow^{(\underline{\mathsf{int}}\to\underline{\mathsf{int}})\to\underline{\mathsf{int}}} (\lambda\varphi.\,\varphi\,(\widehat{LIT}_{\mathsf{int}}\,3))\\
&\quad= \widehat{LAM}\,(\ulcorner\underline{\mathsf{int}}\to\underline{\mathsf{int}}\urcorner, \lambda v^{\mathcal{V}}.\downarrow^{\underline{\mathsf{int}}}\Big((\lambda\varphi.\,\varphi\,(\widehat{LIT}_{\mathsf{int}}\,3))\,(\uparrow_{\underline{\mathsf{int}}\to\underline{\mathsf{int}}}(\widehat{VAR}\,v))\Big))\\
&\quad= \widehat{LAM}\,(ARR(BASE_{\mathsf{int}}, BASE_{\mathsf{int}}), \lambda v^{\mathcal{V}}.\,\widehat{APP}\,(\widehat{VAR}\,v)\,(\widehat{LIT}_{\mathsf{int}}\,3))
\end{aligned}$$

And applying this value to 7 as the first bound-variable index gives us precisely the normal-form term from Example 3 at the end of the previous section.

3.3 The algorithm

So far, we have looked at a semantic property: from the interpretation of a lambda-term in a non-standard denotational semantics of the dynamic signature, we can apparently recover that term's normal form. But this semantic result also forms the basis of an eminently practical normalization *algorithm*, obtained by pulling back the components of the residualizing semantics to the level of program syntax.

We say that a *realization* Φ of a signature Σ in a programming language given by Σ_{pl} is a substitution assigning to every type constant of Σ, a type over Σ_{pl}, and to every term constant of Σ, a Σ_{pl}-term. (For simplicity, we assume that the literals of Σ's base types are also literals of the corresponding Σ_{pl}-types.)

Suppose now that $\Sigma_{\mathrm{s}} \subseteq \Sigma_{\mathrm{pl}}$, $\mathcal{I}_{\mathrm{pl}}$ agrees with $\mathcal{I}_{\mathrm{s}}$, and Σ_{pl} also has some distinguished base types typ and exp with $\mathcal{B}_{\mathrm{pl}}(\mathsf{typ}) = T$ and $\mathcal{B}_{\mathrm{pl}}(\mathsf{exp}) = \Lambda$, as well as the associated (strict) constructor constants. Note that $\llbracket\mathsf{int}\to\mathsf{exp}\rrbracket = \hat{\Lambda}$ (with $\mathcal{N} = \mathbf{Z}_\perp$). Then we can realize the base types of $(\Sigma_{\mathrm{s}}, \Sigma_{\mathrm{d}})$ in Σ_{pl} by

$$\Phi^{\mathrm{r}}(\overline{b}) = b \qquad\qquad \Phi^{\mathrm{r}}(\underline{b}) = \mathsf{int}\to\mathsf{exp}$$

so that $\llbracket\sigma\{\Phi^{\mathrm{r}}\}\rrbracket^{\mathcal{B}_{\mathrm{pl}}} = \llbracket\sigma\rrbracket_{\mathrm{r}}$. Further, for any τ, we can define closed Σ_{pl}-terms,

$$\mathit{name}_\tau : \mathsf{typ} \qquad \mathit{reify}_\tau : \tau\{\Phi^{\mathrm{r}}\}\to\mathsf{int}\to\mathsf{exp} \qquad \mathit{reflect}_\tau : (\mathsf{int}\to\mathsf{exp})\to\tau\{\Phi^{\mathrm{r}}\}$$

such that $[\![\mathsf{name}_\tau]\!]^{\mathcal{I}_{\mathrm{pl}}}\emptyset = \mathbf{val}^\perp \ulcorner\tau\urcorner$, $[\![\mathsf{reify}_\tau]\!]^{\mathcal{I}_{\mathrm{pl}}}\emptyset = \downarrow^\tau$, and $[\![\mathsf{reflect}_\tau]\!]^{\mathcal{I}_{\mathrm{pl}}}\emptyset = \uparrow_\tau$. And using those, we can define realizations of the term constants from $(\Sigma_{\mathrm{s}}, \Sigma_{\mathrm{d}})$:

$$\begin{aligned}
\Phi^{\mathrm{r}}(\overline{c}_{\sigma_1,\ldots,\sigma_n}) &= c_{\sigma_1\{\Phi^{\mathrm{r}}\},\ldots,\sigma_n\{\Phi^{\mathrm{r}}\}} \\
\Phi^{\mathrm{r}}(\underline{c}_{\tau_1,\ldots,\tau_n}) &= \mathsf{reflect}_{\Sigma_{\mathrm{d}}(\underline{c}_{\tau_1,\ldots,\tau_n})}\,(\lambda i.\,\mathsf{CST}\,(c, [\mathsf{name}_{\tau_1}, \ldots, \mathsf{name}_{\tau_n}])) \\
\Phi^{\mathrm{r}}(\$_b) &= \lambda n.\,\lambda i.\,\mathsf{LIT}_b\, n \quad (\text{given } \mathcal{C}_{\mathrm{pl}}(\mathsf{LIT}_b) = \lambda x.\,\mathbf{let}^\perp\ n \Leftarrow x \ \mathbf{in}\ \mathbf{val}^\perp\ \mathit{LIT}_b(n))
\end{aligned}$$

so that $[\![E\{\Phi^{\mathrm{r}}\}]\!]^{\mathcal{I}_{\mathrm{pl}}} = [\![E]\!]_{\mathrm{r}}$. Note in particular that the realizations of static base types and constants are exactly the corresponding constructs from the programming language. This means that we can even use the usual syntactic sugar (such as **letrec** for applications of $\overline{\mathsf{fix}}$) in the static parts of programs to be specialized.

We can use this realization to express our normalization algorithm:

Definition 4 (TDPE). *For any dynamic type* τ*, we define the partial function* $\mathit{TDPE}_\tau : \{E \mid\ \vdash_{\Sigma_{\mathrm{s}},\Sigma_{\mathrm{d}}} E : \tau\} \rightharpoonup \{E \mid\ \vdash^{\mathrm{nf}}_{\Sigma_{\mathrm{d}}} E : \tau\}$ *by*

$$\mathit{TDPE}_\tau(E) = \tilde{E} \text{ if } \mathit{Eval}_{\mathcal{I}_{\mathrm{pl}}}\big(\mathsf{reify}_\tau\, E\{\Phi^{\mathrm{r}}\}\, 0\big) = \ulcorner\tilde{E}\urcorner\,.$$

This *TDPE* is clearly computable; we will show in Section 4 that it is indeed a complete static-normalization function.

Note that we can view TDPE an instance of "cogen-based specialization" [13], in which a "compiler generator" is used to syntactically transform a (binding-time annotated) program $\vdash p : S \times D \to R$ into its *generating extension* $\vdash p^\dagger : S \to \mathsf{exp}$, with the property that for any $s : S$, $\mathit{Eval}(p^\dagger\ s) = \ulcorner p_s \urcorner$. That is, we effectively take

$$p^\dagger = \lambda s^S.\,\mathsf{reify}_{D \to R}\,(\lambda d^D.\,p\{\Phi^{\mathrm{r}}\}\,(s, d))\,0\,.$$

TDPE shares the general high efficiency of cogen-based PE [12]. Formulating the task in terms of static normalization over a binding-time separated signature, however, permits a very precise yet concise syntactic characterization of the specialized program p_s. Also, unlike traditional cogens, TDPE does not require any binding-time annotation of lambdas and applications in the source program.

As a further advantage, the signatures and realizations can be very conveniently expressed in terms of parameterized modules in a Standard ML-style module system. The program to be specialized is simply written as the body of a functor parameterized by the signature of dynamic operations. The functor can then be applied to either an evaluating (Φ^{e}) or a residualizing (Φ^{r}) structure. That is, the cogen pass does not even require an explicit syntactic traversal of the program, making it possible to enrich the static fragment of the language (e.g., with pattern matching) without any modification to the partial evaluator itself.

It is also worth noting that the τ-indexed families above can be straightforwardly defined even in ML's type system: consider the type abbreviation

$$\mathsf{tdpe}(\alpha) \equiv \mathsf{typ} \times (\alpha \to \mathsf{int} \to \mathsf{exp}) \times ((\mathsf{int} \to \mathsf{exp}) \to \alpha)\,.$$

Then for any dynamic type τ, we can construct a term of type $\mathsf{tdpe}(\tau\{\Phi^{\mathrm{r}}\})$ whose value is the triple $(\mathsf{name}_\tau, \mathsf{reify}_\tau, \mathsf{reflect}_\tau)$. We do this by defining once and for all two ML-typable terms

$$\mathsf{base} : \mathsf{tdpe}(\mathsf{int} \to \mathsf{exp}) \qquad \mathsf{arrow} : \forall \alpha, \beta.\ \mathsf{tdpe}(\alpha) \times \mathsf{tdpe}(\beta) \to \mathsf{tdpe}(\alpha \to \beta)\,,$$

with which we can then systematically construct the required value. The technique is explained in more detail elsewhere [18].

Finally, the dynamic polymorphic constants (e.g., $\underline{\mathsf{fix}}$) now take explicit representations of the types at which they are being instantiated as extra arguments. In the evaluating realization, these extra arguments are ignored; but the residualizing realization uses them to construct the name-reflect-reify triple for $\Sigma(\underline{c}_{\tau_1,\ldots,\tau_n})$ given corresponding triples for $\tau_1, \ldots, \tau_n$.

3.4 Applications

Despite its apparent simplicity, TDPE has been successfully used for several nontrivial examples; see Danvy's tutorial for an overview [6]. Many of these actually use the slightly more complicated call-by-value version [4] (see Section 5.4). Because it exploits the highly-optimized evaluation mechanism of a functional language, such a partial evaluator is typically much faster than one representing and manipulating the source program as an explicit value.

Let us just mention here that in addition to stand-alone, source-to-source PE, the TDPE framework can be particularly naturally employed as a "semantic backend" for executable language specifications. That is, if we explicitly parameterize such a specification by the signature of runtime operations (including conditionals, fixed points, etc.), we can instantiate this signature with either the runtime realization, yielding an interpreter, or with the residualizing signature, yielding a compiler [8, 11]. Amusingly, the specializer does not even need the actual *text* of the specification, only its representation as an already compiled module.

4 Showing Correctness

In this section, we sketch a correctness proof for the TDPE algorithm, i.e., that it computes static normal forms when they exist. For the case without static constants, essentially the same algorithm can actually be extracted directly from the standard (syntactic) proof of strong normalization for the simply typed lambda-calculus [1]; but it is not clear if this approach can be extended to a richer programming-language setting.

Instead, our proof uses the technique of semantic logical relations, structured similarly to Gomard and Jones's proof of Lambda-mix [14, 8.8], but accounting more rigorously for potential divergence and for the generation of "fresh" variable names. It also admits a richer type structure for the dynamic language. (We can still treat the untyped variant as a special case; see Section 5.1.)

4.1 Properties of the term-family representation

Let some evaluating dynamic interpretation $\mathcal{I}_\mathrm{d}^\mathrm{e}$ of Σ_d be given. We abbreviate $[\![-]\!]^{\mathcal{I}_\mathrm{s},\mathcal{I}_\mathrm{d}^\mathrm{e}}$ as $[\![-]\!]_\mathrm{e}$ (and $[\![-]\!]^{\mathcal{I}_\mathrm{s},\mathcal{I}_\mathrm{d}^\mathrm{r}}$ as $[\![-]\!]_\mathrm{r}$ as before).

Definition 5 (partial meaning relation, $\succ$). *For any Δ, let*

$$\sharp\Delta = \max(\{i+1 \mid \mathrm{g}_i \in \operatorname{dom}\Delta\} \cup \{0\})$$

(so if $i \geq \sharp\Delta$ then $\mathrm{g}_i \notin \operatorname{dom}\Delta$). Then for any Δ, τ, $s \in \{\mathrm{nf}, \mathrm{at}\}$ (as used in Definition 2), $\delta \in \llbracket\Delta\rrbracket^{\mathcal{I}_\mathrm{d}^\mathrm{e}}$, $e \in \hat{\Lambda}$, and $a \in \llbracket\tau\rrbracket^{\mathcal{I}_\mathrm{d}^\mathrm{e}}$, we define a relation by

$$e \mathbin{@^{\Delta}} \delta \succ_{\tau}^{s} a \iff \forall i \geq \sharp\Delta.\, e\, i = \bot \vee \exists E.\, e\, i = \mathbf{val}^{\bot}\, \ulcorner E \urcorner \wedge \Delta \vdash_{\Sigma_\mathrm{d}}^{s} E : \tau \wedge \llbracket E\rrbracket^{\mathcal{I}_\mathrm{d}^\mathrm{e}}\delta = a$$

This roughly expresses that "$\llbracket e\rrbracket\delta = a$", but taking into account variable renaming, partiality, and simplification: for all sufficiently large starting indices i, if $e\, i$ converges, it must represent a normal-form term with the right meaning. We check that this relation is semantically well behaved:

Definition 6 (admissibility). *We say that a relation $R \subseteq A \times A'$ between two pointed cpos is* admissible *(or* inclusive*) if it is chain-complete (i.e., for all chains $(a_i)_i$ and $(a'_i)_i$, if $\forall i.\, (a_i, a'_i) \in R$ then also $(\bigsqcup_i a_i, \bigsqcup_i a'_i) \in R$) and* pointed *(i.e., $(\bot_A, \bot_{A'}) \in R$).*

Lemma 1 ($\succ$ is admissible). *For any Δ, $\delta \in \llbracket\Delta\rrbracket^{\mathcal{I}_\mathrm{d}^\mathrm{e}}$, τ, and $s \in \{\mathrm{nf}, \mathrm{at}\}$, the relation $\{(e, a) \mid e \mathbin{@^{\Delta}} \delta \succ_{\tau}^{s} a\}$ is admissible.*

Proof. Straightforward, noting that admissible relations are closed under arbitrary intersection, and that any chain in $\Lambda_\bot$ is eventually constant. □

Although the output of TDPE is a closed program, we still need to account for the typing and meaning of open program fragments as they are being constructed and put into context:

Definition 7 (Kripke structure). *A* world *is a pair (Δ, δ) where $\delta \in \llbracket\Delta\rrbracket^{\mathcal{I}_\mathrm{d}^\mathrm{e}}$. Such worlds are partially ordered by*

$$(\Delta', \delta') \geq (\Delta, \delta) \iff \forall x \in \operatorname{dom}\Delta.\, \Delta'(x) = \Delta(x) \wedge \delta'\, x = \delta\, x$$

Lemma 2 ($\succ$ is Kripke). *If $e \mathbin{@^{\Delta}} \delta \succ_{\tau}^{s} a$ and $(\Delta', \delta') \geq (\Delta, \delta)$, then also $e \mathbin{@^{\Delta'}} \delta' \succ_{\tau}^{s} a$.*

Proof. Follows easily from standard weakening properties of the typing relation (if $\Delta \vdash E : \tau$ then $\Delta' \vdash E : \tau$) and denotational semantics ($\llbracket E\rrbracket\,\delta' = \llbracket E\rrbracket\,\delta$). □

Lemma 3 (meanings of term families). *The wrapper functions have the following properties:*

1. *If $\Delta(v) = \tau$ then $\widehat{VAR}\, v \mathbin{@^{\Delta}} \delta \succ_{\tau}^{\mathrm{at}} \delta\, v$.*
2. *If $n \in \mathcal{B}_\mathrm{s}(b)$ then $\widehat{LIT}_b\, n \mathbin{@^{\Delta}} \delta \succ_{\underline{b}}^{\mathrm{at}} \mathbf{val}^{\bot}\, n$*
3. *$\widehat{CST}\,(c, [\ulcorner\tau_1\urcorner, \ldots, \ulcorner\tau_n\urcorner]) \mathbin{@^{\Delta}} \delta \succ_{\Sigma_\mathrm{d}(\underline{c}_{\tau_1,\ldots,\tau_n})}^{\mathrm{at}} \mathcal{C}_\mathrm{d}^\mathrm{e}(\underline{c}_{\tau_1,\ldots,\tau_n})$*
4. *If for all $v \notin \operatorname{dom}\Delta$ and $a \in \llbracket\tau_1\rrbracket^{\mathcal{I}_\mathrm{d}^\mathrm{e}}$, $\varepsilon\, v \mathbin{@^{\Delta, v:\tau_1}} \delta[v \mapsto a] \succ_{\tau_2}^{\mathrm{nf}} f\, a$*
   then $\widehat{LAM}\,(\ulcorner\tau_1\urcorner, \varepsilon) \mathbin{@^{\Delta}} \delta \succ_{\tau_1 \to \tau_2}^{\mathrm{nf}} f$.
5. *If $e_1 \mathbin{@^{\Delta}} \delta \succ_{\tau_1 \to \tau_2}^{\mathrm{at}} f$ and $e_2 \mathbin{@^{\Delta}} \delta \succ_{\tau_1}^{\mathrm{nf}} a$ then $\widehat{APP}\,(e_1, e_2) \mathbin{@^{\Delta}} \delta \succ_{\tau_2}^{\mathrm{at}} f\, a$*

Proof. Straightforward verification in all cases. (For case 4, we exploit the fact that $\llbracket\tau_1\rrbracket^{\mathcal{I}_\mathrm{d}^\mathrm{e}}$ is always non-empty; this shortcut can be avoided by using a slightly more complicated world structure throughout the proof.) □

4.2 Soundness of TDPE

We prove soundness by formally relating the standard and the residualizing interpretations of types and terms.

Definition 8 (logical relation, $\sim_\sigma$). *For any type σ and world (Δ, δ), we define a relation $a \mathbin{@^{\Delta}} \delta \sim_\sigma a'$, where $a \in \llbracket \sigma \rrbracket_{\mathrm{r}}$ and $a' \in \llbracket \sigma \rrbracket_{\mathrm{e}}$ by:*

$$
\begin{aligned}
n \mathbin{@^{\Delta}} \delta \sim_{\overline{b}} n' &\iff n = n' \\
e \mathbin{@^{\Delta}} \delta \sim_{\underline{b}} n' &\iff e \mathbin{@^{\Delta}} \delta \succ^{\mathrm{nf}}_{\underline{b}} n' \\
f \mathbin{@^{\Delta}} \delta \sim_{\sigma_1 \to \sigma_2} f' &\iff \forall (\Delta', \delta') \geq (\Delta, \delta). \\
&\qquad\qquad \forall a, a'.\, a \mathbin{@^{\Delta'}} \delta' \sim_{\sigma_1} a' \Rightarrow f\,a \mathbin{@^{\Delta'}} \delta' \sim_{\sigma_2} f'\,a'
\end{aligned}
$$

We first check the standard requirements:

Lemma 4 ($\sim_\sigma$ is admissible). *For any σ, the relation $\{(a, a') \mid a \mathbin{@^{\Delta}} \delta \sim_\sigma a'\}$ is admissible.*

Proof. Simple induction on σ. For $\underline{b}$, use admissibility of $\succ$ (Lemma 1). □

Lemma 5 ($\sim_\sigma$ is Kripke). *If $e \mathbin{@^{\Delta}} \delta \sim_\sigma a$ and $(\Delta', \delta') \geq (\Delta, \delta)$, then also $e \mathbin{@^{\Delta'}} \delta' \sim_\sigma a$.*

Proof. This is a standard result about Kripke logical relations; the proof is by a simple induction on σ, using Lemma 2 for the base case $\sigma = \underline{b}$. □

We obtain our main correctness result from two lemmas:

Lemma 6 (soundness, type part). *For any dynamic type τ,*

1. *If $a \mathbin{@^{\Delta}} \delta \sim_\tau a'$ then $\downarrow^{\tau} a \mathbin{@^{\Delta}} \delta \succ^{\mathrm{nf}}_{\tau} a'$.*
2. *If $e \mathbin{@^{\Delta}} \delta \succ^{\mathrm{at}}_{\tau} a'$ then $\uparrow_{\tau} e \mathbin{@^{\Delta}} \delta \sim_\tau a'$.*

Proof. Straightforward induction on τ, using the properties of the wrapper functions from Lemma 3. □

Lemma 7 (soundness, term part). *Let (Δ, δ) be a world, and let $\rho \in \llbracket \Gamma \rrbracket_{\mathrm{r}}$ and $\rho' \in \llbracket \Gamma \rrbracket_{\mathrm{e}}$. Then for any well-typed term $\Gamma \vdash_{\Sigma_{\mathrm{s}}, \Sigma_{\mathrm{d}}} E : \sigma$, if $\forall x \in \mathrm{dom}\, \Gamma.\, \rho x \mathbin{@^{\Delta}} \delta \sim_{\Gamma(x)} \rho' x$ then $\llbracket E \rrbracket_{\mathrm{r}} \rho \mathbin{@^{\Delta}} \delta \sim_\sigma \llbracket E \rrbracket_{\mathrm{e}} \rho'$.*

Proof. This is the usual Kripke logical relations lemma, proved by straightforward induction on E. The only non-standard case is that of $E = \underline{c}_{\tau_1, \ldots, \tau_n}$, for which we need Lemma 6(2). For $E = \overline{\mathsf{fix}}_\sigma$, we use fixed-point induction, i.e., Lemma 4 together with the chain-based construction of $\mathcal{C}_{\mathrm{s}}(\mathsf{fix}_\sigma)$ in Definition 1. □

Theorem 1 (soundness). *TDPE is a static-normalization function.*

Proof. Observe first that $TDPE_\tau(E) = \tilde{E}$ iff $\downarrow^{\tau} (\llbracket E \rrbracket_{\mathrm{r}}\, \emptyset)\, 0 = \mathbf{val}^{\perp} \ulcorner \tilde{E} \urcorner$. Now, by Lemma 7, since empty environments are vacuously related, $\llbracket E \rrbracket_{\mathrm{r}}\, \emptyset \mathbin{@} \emptyset \sim_\tau \llbracket E \rrbracket_{\mathrm{e}}\, \emptyset$. And thus by Lemma 6(1), $\downarrow^{\tau} (\llbracket E \rrbracket_{\mathrm{r}}\, \emptyset) \mathbin{@} \emptyset \succ^{\mathrm{nf}}_{\tau} \llbracket E \rrbracket_{\mathrm{e}}\, \emptyset$, which, by the definitions of $\succ_\tau$ and $\sharp$, gives us precisely that $\vdash^{\mathrm{nf}}_{\Sigma_{\mathrm{d}}} \tilde{E} : \tau$ and $\llbracket \tilde{E} \rrbracket^{\mathcal{I}^{\mathrm{e}}_{\mathrm{d}}} = \llbracket E \rrbracket^{\mathcal{I}_{\mathrm{s}}, \mathcal{I}^{\mathrm{e}}_{\mathrm{d}}}$.

For the second part, if $E' =^{\mathcal{I}_{\mathrm{s}}} E$ then in particular $\llbracket E' \rrbracket_{\mathrm{r}} = \llbracket E \rrbracket_{\mathrm{r}}$. And thus by the observation above, we must have $TDPE_\tau(E') = TDPE_\tau(E)$. □

4.3 Completeness of TDPE

To supplement the above partial-correctness result, we can also show that if a suitable $\tilde{E}$ exists, the algorithm will actually find it. This proof uses a much simpler logical-relation argument, capturing the intuition that the algorithm necessarily converges when applied to a term containing no static constants:

Definition 9 (totality predicate). *For any dynamic type* τ*, we define a predicate* $T_\tau \subseteq \llbracket \tau \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}}$ *by*

$$T_{\underline{b}} = \{ e \in \hat{\Lambda} \mid \forall i.\, ei \neq \bot \}$$

$$T_{\tau_1 \to \tau_2} = \{ f \in \llbracket \tau_1 \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}} \to \llbracket \tau_2 \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}} \mid \forall a \in T_{\tau_1}.\, f a \in T_{\tau_2} \}$$

As before, we then obtain the result from two main lemmas:

Lemma 8 (completeness, type part). *For any dynamic type* τ*,*

1. *If* $a \in T_\tau$ *then for all* $i \geq 0$*,* $\downarrow^\tau a\, i \neq \bot$*.*
2. *If for all* $i \geq 0$*,* $e\, i \neq \bot$ *then* $\uparrow_\tau e \in T_\tau$*.*

Proof. Straightforward induction on τ, by inspection of the definitions of the wrapper functions. □

Lemma 9 (completeness, term part). *Let* $\delta \in \llbracket \Delta \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}}$*. Then for any well-typed term* $\Delta \vdash_{\Sigma_{\mathrm{d}}} E : \tau$*, if* $\forall x \in \mathrm{dom}\, \Delta.\, \delta x \in T_{\Delta(x)}$ *then* $\llbracket E \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}} \delta \in T_\tau$*.*

Proof. Standard induction on E, using Lemma 8(2) for dynamic constants. □

Theorem 2 (completeness). *TDPE is a complete static-normalization function.*

Proof. Suppose $\vdash_{\Sigma_{\mathrm{s}}, \Sigma_{\mathrm{d}}} E : \tau$ has the static normal form $\vdash_{\Sigma_{\mathrm{d}}} \tilde{E} : \tau$. Then in particular $\llbracket E \rrbracket_{\mathrm{r}} = \llbracket E \rrbracket^{\mathcal{I}_{\mathrm{s}}, \mathcal{I}_{\mathrm{d}}^{\mathrm{r}}} = \llbracket \tilde{E} \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}}$. By Lemma 9, $\llbracket \tilde{E} \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}} \emptyset \in T_\tau$, and thus by Lemma 8(1), $\downarrow^\tau (\llbracket E \rrbracket_{\mathrm{r}}\, \emptyset) 0 = \downarrow^\tau (\llbracket \tilde{E} \rrbracket^{\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}}\, \emptyset) 0 \neq \bot$, so $\mathit{TDPE}_\tau(E)$ is defined. □

5 Variations and Extensions

5.1 Lambda-mix

We can use the previous results to show correctness of partial evaluation for languages like the one used for Lambda-mix [14, 8.8]. Here, the dynamic language is untyped. Or, more precisely, it has a single type $\underline{d}$ of dynamic values, and operators:

$$\frac{\Gamma, x{:}\underline{d} \vdash E : \underline{d}}{\Gamma \vdash \underline{\lambda} x.\, E : \underline{d}} \qquad \frac{\Gamma \vdash E_1 : \underline{d} \qquad \Gamma \vdash E_2 : \underline{d}}{\Gamma \vdash E_1 \mathbin{\underline{@}} E_2 : \underline{d}}$$

To model this in our typed framework, we let the dynamic signature Σ_{d} contain the single base type $\underline{d}$, and constants $\underline{\phi} : (\underline{d} \to \underline{d}) \to \underline{d}$ and $\underline{\psi} : \underline{d} \to \underline{d} \to \underline{d}$. We can then treat dynamic lambda-abstraction and application as abbreviations:

$$\underline{\lambda} x.\, E \equiv \underline{\phi}(\lambda x^{\underline{d}}.\, E) \qquad \text{and} \qquad E_1 \mathbin{\underline{@}} E_2 \equiv \underline{\psi}\, E_1\, E_2$$

In the evaluating dynamic semantics $\mathcal{I}_{\mathrm{d}}^{\mathrm{e}}$ of Σ_{d}, the type constant $\underline{d}$ is interpreted as a solution to the domain equation

$$D \cong (D \rightarrow D) \oplus \mathbf{Z}_{\perp} \oplus \mathbf{B}_{\perp}\,,$$

and $\underline{\phi}$ and $\underline{\psi}$ as the evident embedding and projection functions for the first summand. In the residualizing interpretation $\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}$, on the other hand, $\underline{d}$ is the domain of syntactic term families, and the constants are reflected according to their types, as usual. In particular, $\mathit{reify}_{\underline{d}}$ is simply the identity.

The soundness result for TDPE then gives us that if $\vdash_{\Sigma_{\mathrm{s}},\Sigma_{\mathrm{d}}} E : \underline{d}$ and $\mathit{Eval}_{\mathcal{I}_{\mathrm{s}},\mathcal{I}_{\mathrm{d}}^{\mathrm{r}}}(E) = \ulcorner \tilde{E} \urcorner$ then $\vdash^{\mathrm{nf}}_{\Sigma_{\mathrm{d}}} \tilde{E} : \underline{d}$ and $\mathit{Eval}_{\mathcal{I}_{\mathrm{d}}^{\mathrm{e}}}(\tilde{E}) = \mathit{Eval}_{\mathcal{I}_{\mathrm{s}},\mathcal{I}_{\mathrm{d}}^{\mathrm{e}}}(E)$. With a little more work we also obtain a similar statement for non-closed terms.

Note finally that normalizing a term of type $\underline{d}$ a priori yields a simply-typed term over Σ_{d}, rather than an untyped one. In a static normal form, however, any occurrence of the constant $\underline{\phi}$ will be applied to a syntactic lambda-abstraction, and $\underline{\psi}$ will be applied to two arguments. Thus, the output of the partial evaluator can always be directly expressed in terms of the underlined abstraction and application operators.

5.2 Gensym-like name generation

The term-family representation from Section 3.1 constructs terms in which the names of bound variables are derived from the number of enclosing lambdas; this convention is sometimes known as *de Bruijn levels* (not to be confused with de Bruijn *indices*). Although this is probably the simplest choice in a purely functional setting, there is nothing canonical about it. To more precisely capture the informal concept of newly-generated "fresh" variable names, we could instead take:

$$\begin{aligned}
\hat{\Lambda} &= \mathcal{N} \rightarrow (\Lambda \times \mathcal{N})_{\perp} \\[1ex]
\widehat{\mathit{VAR}} &= \lambda v.\, \lambda i.\, \mathbf{val}^{\perp}\, (\mathit{VAR}(v), i) \\
\widehat{\mathit{LAM}} &= \lambda(t, \varepsilon).\, \lambda i.\, \mathbf{let}^{\perp}\ (l, i') \Leftarrow \varepsilon\, \mathrm{g}_i\, (i+1)\ \mathbf{in}\ \mathbf{val}^{\perp}\, (\mathit{LAM}(\mathrm{g}_i, t, l), i') \\
\widehat{\mathit{APP}} &= \lambda(e_1, e_2).\, \lambda i.\, \mathbf{let}^{\perp}\ (l_1, i') \Leftarrow e_1\, i\ \mathbf{in}\ \mathbf{let}^{\perp}\ (l_2, i'') \Leftarrow e_2\, i'\ \mathbf{in}\ \mathbf{val}^{\perp}\, (\mathit{APP}(l_1, l_2), i'')
\end{aligned}$$

(with analogous extensions for constants and literals). This scheme generates terms in which all bound-variable names are distinct. Then, after changing the conditional meaning relation to read:

$$\begin{aligned}
&e \mathbin{@^{\Delta}} \delta \succ^{s}_{\tau} a \iff \\
&\qquad \forall i \geq \sharp\Delta.\, e\, i = \perp \vee \exists E, i' \geq i.\, e\, i = \mathbf{val}^{\perp}\, (\ulcorner E \urcorner, i') \wedge \Delta \vdash^{s}_{\Sigma_{\mathrm{d}}} E : \tau \wedge [\![E]\!]\delta = a.
\end{aligned}$$

we can check that Lemma 3 still holds, and therefore all the remaining constructions and proofs go through without further modifications.

5.3 On-line type-directed partial evaluation

Although TDPE generally works on binding-time separated signatures, it is actually possible to give an on-line formulation, in which it is not necessary to explicitly annotate all base types and operations. Conceptually, we instead take $\mathcal{B}^{\mathrm{r}}_{\mathrm{d}}(b) = \mathcal{N} \to (\mathcal{B}_{\mathrm{s}}(b) + \Lambda)_{\perp}$. (In practice, when Λ is a conveniently inspectable type, it suffices to take $\mathcal{B}^{\mathrm{r}}_{\mathrm{d}}(b) = \hat{\Lambda}$ and explicitly recognize Λ-values that represent literals.) The arithmetic operators then produce a static result if if both arguments are static, and dynamic otherwise, possibly coercing one argument in the process. A similar extension works for conditionals.

This scheme enables "opportunistic" simplifications, in cases where an operand is sometimes, but not always, statically known (or where a static analysis cannot prove that it is known). Note, however, that we must still annotate occurrences of fix as static or dynamic, or otherwise prevent fruitless infinite expansion of a recursive function. For example, it is often possible to explicitly identify a particular function parameter as the one controlling the recursion, and only unfold calls in which that argument is a literal [5].

Of course, the price of these potential improvements is that the amount of simplification is less predictable: the output will still be in long $\beta\eta$-normal form, but it is no longer evident from the original program which operations will be performed statically, and which ones must remain in the specialized program.

5.4 Call-by-value and effects

For practical applications, a call-by-value variant of TDPE is usually preferable, and indeed the technique was first presented in this setting [4]. Let us briefly sketch the necessary changes from the call-by-name case here.

To give a denotational semantics of an ML-like language, we consider an interpretation $\mathcal{I}$ to also explicitly include a *monad* for modeling effectful computations. We usually want to be able to use any monad in the evaluating interpretation; thus the notion of static equality must be safe for any dynamic effect. That is, instead of computing normal forms based on the strong $\beta\eta$-lambda-calculus, we now need a normalization-by-evaluation algorithm for Moggi's computational lambda-calculus λ_c [15].

Fortunately, much as a single residualizing interpretation of dynamic type and term constants suffices to compute call-by-name static normal forms sound for any dynamic interpretation, it turns out that a single "maximally general" residualizing interpretation of effects can be used to compute call-by-value normal forms suitable for any dynamic monad.

A particularly natural such residualizing monad is that of continuations with answer type $\hat{\Lambda}$, which can be straightforwardly related to any dynamic monad for the purpose of the logical relation in Section 4.2. Moreover, we can still construct the corresponding residualizing realization Φ^{r}, as long as our programming language contains Scheme-style first-class continuations and state [10]. Incidentally, this construction also allows disjoint-union types (sums) to be naturally added to the language. The details are still under investigation, however, and will be reported in a forthcoming paper.

6 Conclusions and Future Work

We have given an account of type-directed partial evaluation that separates the specification of the problem (computation of static normal forms) from its implementation (normalization by evaluation); in previous work these tended to be intertwined. We also presented a correctness proof for the implementation, using logical relations over a simple denotational model of the binding-time separated language. To keep the details manageable, we restricted our scope to a purely functional language; but both the algorithm and the proof techniques extend to call-by-value languages with effects as well.

Future work falls in two classes. First, there are a number of natural extensions to the framework and results in essentially the form they are presented here. In addition to the directions already mentioned in Section 5, one can also consider polyvariant specialization, run-time code generation, and other classical PE concepts in the context of TDPE.

Second, it would be interesting to investigate how TDPE relates to more general work on linguistic support for staged computation, especially recent developments based on modal logics [9, 16]. For example, it might be possible to generalize the notion of static normalization to such settings, and consider normalization-by-evaluation algorithms for type systems more expressive than simple types.

Acknowledgments

I want to thank Olivier Danvy for many deep and fruitful discussions about TDPE. I am also grateful to Daniel Damian, Bernd Grobauer, Zhe Yang, and the PPDP'99 reviewers for their careful reading and insightful comments on various drafts of this manuscript.

References

1. Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993.
2. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991.
3. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
4. Olivier Danvy. Type-directed partial evaluation. In *23rd ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996.

5. Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295, Kyoto, Japan, April 1998.
6. Olivier Danvy. Type-directed partial evaluation. Lecture Notes BRICS LN-98-3, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998. To appear in LNCS.
7. Olivier Danvy and Peter Dybjer, editors. *Preliminary Proceedings of the APPSEM Workshop on Normalization by Evaluation*, Chalmers, Sweeden, May 1998. BRICS Note NS-98-1.
8. Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996.
9. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd ACM Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg Beach, Florida, January 1996.
10. Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.
11. William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 122–131, Chicago, Illinois, May 1998. IEEE Computer Society.
12. Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In J. Hsiang and A. Ohori, editors, *Advances in Computing Science – ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manilla, The Philippines, December 1998.
13. Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991.
14. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.
15. Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.
16. Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler and more expressive. In *8th European Symposium on Programming Languages and Systems*, number 1576 in Lecture Notes in Computer Science, pages 193–207, Amsterdam, The Netherlands, March 1999.
17. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.
18. Zhe Yang. Encoding types in ML-like languages. In *International Conference on Functional Programming*, pages 289–300, Baltimore, Maryland, September 1998.