Normalization by Evaluation for the Computational Lambda-Calculus

To appear, Typed Lambda Calculi and Applications 2001

Andrzej Filinski

BRICS*, Department of Computer Science, University of Aarhus
`andrzej@brics.dk`

Abstract. We show how a simple semantic characterization of normalization by evaluation for the $\lambda_{\beta\eta}$-calculus can be extended to a similar construction for normalization of terms in the computational λ-calculus. Specifically, we show that a suitable *residualizing* interpretation of base types, constants, and computational effects allows us to extract a syntactic normal form from a term's denotation. The required interpretation can itself be constructed as the meaning of a suitable functional program in an ML-like language, leading directly to a practical normalization algorithm. The results extend easily to product and sum types, and can be seen as a formal basis for call-by-value type-directed partial evaluation.

1 Introduction

The basic idea of normalization by evaluation is to extract the normal form (with respect to some notion of conversion) of a term from its interpretation in a suitably chosen, quasi-syntactic denotational model of the conversion relation [5].

For instance, let us consider the interpretation of a pure, simply typed lambda-term E in a model where all base types are interpreted as the set Λ of well-formed lambda-terms, and function types are interpreted as full set-theoretic function spaces. Then it is fairly simple to (at least informally) construct for any type τ, a function $\mathit{nf}_\tau \in [\![\tau]\!] \to \Lambda$, such that for any closed term $\widetilde{E} : \tau$ in $\beta\eta$-long normal form, $\mathit{nf}_\tau([\![\widetilde{E}]\!]) =_\alpha \widetilde{E}$. We proceed as follows:

Let $\tau = \tau_1 \to \cdots \to \tau_n \to b$ $(n \geq 0)$, where each $\tau_i = \tau_{i1} \to \cdots \to \tau_{im_i} \to b_i$. Then $\widetilde{E} : \tau$ must be of the form $\lambda x_1. \cdots . \lambda x_n. x_i\, \widetilde{E}_1 \cdots \widetilde{E}_{m_i}$ where each $\widetilde{E}_j : \tau_{ij}$ is again in normal form. We can thus define nf_τ inductively as:

$$\begin{aligned}\mathit{nf}_\tau = \lambda f.\, & \mathit{LAM}(v_1, \ldots, \mathit{LAM}(v_n, \\ & \quad f\,(\lambda a_1. \cdots . \lambda a_{m_1}.\, \mathit{APP}\,(\cdots \mathit{APP}\,(\mathit{VAR}\, v_1, \mathit{nf}_{\tau_{11}}\, a_1) \cdots, \mathit{nf}_{\tau_{1m_1}}\, a_{m_1})) \\ & \qquad \vdots \\ & \quad (\lambda a_1. \cdots . \lambda a_{m_n}.\, \mathit{APP}\,(\cdots \mathit{APP}\,(\mathit{VAR}\, v_n, \mathit{nf}_{\tau_{n1}}\, a_1) \cdots, \mathit{nf}_{\tau_{nm_n}}\, a_{m_n}))))\end{aligned}$$

where the v_i are "fresh" variable names, and we use VAR, LAM, and APP for constructing elements of Λ, to distinguish them from function abstraction and application in the set-theoretic model.

---

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

Moreover, it is easy to see that $\mathit{nf}_\tau(\llbracket - \rrbracket)$ is in fact a normalization function: since $\beta\eta$-convertibility is sound for equality in all set-theoretic interpretations (and hence also in our chosen one), we have, for all terms E and E', that $E =_{\beta\eta} E'$ implies $\llbracket E \rrbracket = \llbracket E' \rrbracket$. So if $\widetilde{E}$ is the $\beta\eta$-long normal form of E, then $\mathit{nf}_\tau(\llbracket E \rrbracket) = \mathit{nf}_\tau(\llbracket \widetilde{E} \rrbracket) =_\alpha \widetilde{E}$.

Finally, if we can also construct a syntactic term $\mathsf{nf}_\tau : \tau \to \Lambda$ such that $\mathit{nf}_\tau \llbracket E \rrbracket = \llbracket \mathsf{nf}_\tau E \rrbracket$, then $\mathsf{nf}_\tau\, E$ is a closed term of base type Λ, and can thus be executed as a functional program. This gives us a very efficient executable *algorithm* for computing normal forms, and was indeed one of the motivations behind the construction [4, 3]: we are reducing the general problem of term normalization to a special case for which we already have a good solution.

A natural question arises whether this semantic technique for normalization of lambda-terms is inherently tied to $\beta\eta$-conversion. Somewhat surprisingly, it is not: in the following, we show how the same idea can be used to normalize terms with respect to the computational lambda-calculus [16], where it also extends to product and – more notably – sum types. In fact, we can systematically extract the computational normal form of any pure, typed lambda-term from only its observable behavior in an imperative functional language such as ML.

Despite the relative simplicity of the construction, there are still a few technical details to nail down, even in the purely functional case. Accordingly, we will first present in Section 2 the normalization algorithm for a call-by-name setting, then show in Section 3 how it can be refined to call-by-value. In Section 4 we show how to further extend the normalizer with product and sum types, and in Section 5 we consider the relationship between normalization by evaluation and type-directed partial evaluation. Finally, Section 6 concludes and points out some directions for further work.

2 Normalization by evaluation for call by name

The normalization construction sketched above, essentially due to Berger and Schwichtenberg [4], has been studied in many formulations [8], including more syntactic variants [2] as well as category-theoretic ones [1, 6]. In the following, we present it in a call-by-name functional-programming setting [11]; this formulation extends particularly naturally to the call-by-value variant in the next section.

2.1 Language and semantic framework

Syntax. A *signature* Σ includes, first, a collection of base types b. The set of well-formed Σ-types τ is then given by the grammar

$$\tau ::= b \mid \tau_1 \to \tau_2$$

Further, Σ assigns Σ-types to a (possibly infinite) collection of constants c. Let x range over variable names, and Γ be a finite assignment of Σ-types to variables. Then the set of well-typed Σ-terms $\Gamma \vdash_\Sigma E : \tau$ is again given by the usual rules:

$$\frac{\Sigma(c) = \tau}{\Gamma \vdash_\Sigma c : \tau} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash_\Sigma E : \tau_2}{\Gamma \vdash_\Sigma \lambda x^{\tau_1}.E : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma E_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_\Sigma E_2 : \tau_1}{\Gamma \vdash_\Sigma E_1\, E_2 : \tau_2}$$

Finally, a Σ-*program* is a closed Σ-term of base type.

Semantics. For concreteness, and to accommodate the refinements in Section 5, we consider only a specific, domain-theoretic framework, but the results also adapt easily to a set-theoretic setting, by forgetting the order structure.

We work in the setting of (bottomless) cpos and (total) continuous functions. We also use the concept of a *monad* (more precisely, a *Kleisli triple*) $\mathcal{T} = (T, \eta, \star)$, where T maps cpos to cpos, $\eta_A \in A \to TA$ is the *unit* function, and $\star_{A,B} \in TA \times (A \to TB) \to TB$ is the *extension* operation written backwards, i.e., with the function last; this makes longer sequences of extensions easier to read. We omit the subscripts on units and extensions where they are clear from the context. A particularly important instance is the *lifting monad*, $\mathcal{T}_{\mathrm{l}}$, where $T^{\mathrm{l}} A = A_\perp = \{\iota\, a \mid a \in A\} \cup \{\perp\}$ with the usual ordering, $\eta^{\mathrm{l}}\, a = \iota\, a$, $\perp \star^{\mathrm{l}} f = \perp$, and $(\iota\, a) \star^{\mathrm{l}} f = f\, a$.

An *interpretation* $\mathcal{I}$ of a signature Σ is a pair of functions $(\mathcal{B}, \mathcal{C})$. $\mathcal{B}$ assigns to every base type b in Σ, a cpo $\mathcal{B}(b)$. This assignment determines for any Σ-type τ, a pointed (i.e., containing a least element) cpo $[\![\tau]\!]^{\mathcal{I}}$ as follows:

$$[\![b]\!]^{\mathcal{I}} = T^{\mathrm{l}}\mathcal{B}(b) \qquad\qquad [\![\tau_1 \to \tau_2]\!]^{\mathcal{I}} = [\![\tau_1]\!]^{\mathcal{I}} \to [\![\tau_2]\!]^{\mathcal{I}}$$

where $A \to B$ denotes the cpo of all continuous functions between A and B. We also give meaning to a type assignment Γ as a finite product:

$$[\![\Gamma]\!]^{\mathcal{I}} = \textstyle\prod_{x \in \mathrm{dom}\, \Gamma} [\![\Gamma(x)]\!]^{\mathcal{I}} = \{\rho \mid \forall x \in \mathrm{dom}\, \Gamma.\, \rho\, x \in [\![\Gamma(x)]\!]^{\mathcal{I}}\}$$

The function $\mathcal{C}$ assigns to every Σ-constant c an element $\mathcal{C}(c) \in [\![\Sigma(c)]\!]^{\mathcal{I}}$. Again this assignment extends to a full semantics of terms: for any $\Gamma \vdash_\Sigma E : \tau$, we define a continuous function $[\![E]\!]^{\mathcal{I}} \in [\![\Gamma]\!]^{\mathcal{I}} \to [\![\tau]\!]^{\mathcal{I}}$ in the usual way:

$$\begin{array}{ll} [\![c]\!]^{\mathcal{I}}\rho = \mathcal{C}(c) & \quad [\![\lambda x^{\tau}.\, E]\!]^{\mathcal{I}}\rho = \lambda a.\, [\![E]\!]^{\mathcal{I}}(\rho[x \mapsto a]) \\ [\![x]\!]^{\mathcal{I}}\rho = \rho\, x & \quad [\![E_1\, E_2]\!]^{\mathcal{I}}\rho = [\![E_1]\!]^{\mathcal{I}}\rho\,([\![E_2]\!]^{\mathcal{I}}\rho) \end{array}$$

Equivalence and normal forms. We say that two Σ-terms E and E' are semantically equivalent, written $\models E = E'$, if for all interpretations $\mathcal{I}$ of Σ, $[\![E]\!]^{\mathcal{I}} = [\![E']\!]^{\mathcal{I}}$. It is easy to see that if $E =_{\beta\eta} E'$ then $\models E = E'$. More generally, if *Int* is a subset of all possible interpretations of Σ (e.g., constraining the meanings of some of the constants), we write $\models^{\mathit{Int}} E = E'$ iff for all $\mathcal{I} \in \mathit{Int}$, $[\![E]\!]^{\mathcal{I}} = [\![E']\!]^{\mathcal{I}}$; we will return to this notion in Section 5.

Among the well-typed terms $\Gamma \vdash_\Sigma E : \tau$, we distinguish those in *normal* and *atomic* (also known as *neutral*) form:

$$\frac{\Gamma \vdash^{\mathrm{at}} E : b}{\Gamma \vdash^{\mathrm{nf}} E : b} \qquad\qquad \frac{\Gamma, x{:}\tau_1 \vdash^{\mathrm{nf}} E : \tau_2}{\Gamma \vdash^{\mathrm{nf}} \lambda x^{\tau_1}.\, E : \tau_1 \to \tau_2}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash^{\mathrm{at}} x : \tau} \qquad \frac{\Sigma(c) = \tau}{\Gamma \vdash^{\mathrm{at}} c : \tau} \qquad \frac{\Gamma \vdash^{\mathrm{at}} E_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash^{\mathrm{nf}} E_2 : \tau_1}{\Gamma \vdash^{\mathrm{at}} E_1\, E_2 : \tau_2}$$

A *normalization function*, in the sense of Coquand and Dybjer [5] (but with a semantic notion of equivalence), then maps any term E to a normal-form term $\mathit{norm}(E)$, such that $\models \mathit{norm}(E) = E$, and such that for all E' with $\models E' = E$, $\mathit{norm}(E') = \mathit{norm}(E)$.

2.2 A normalization result

The traditional way of computing $norm(E)$ is by repeated β-reductions, possibly followed by η-expansions. However, we can also compute $norm$ by a subtler, semantic method, *reduction-free normalization.*

Representing lambda-terms. Let $\mathbf{V}$ be a set (= discrete cpo) of explicitly typed variable names, and $\mathbf{E}$ a set suitable for representing lambda-terms, i.e., allowing us to define injective functions with mutually disjoint ranges,

$$CST \in \mathrm{dom}\,\Sigma \to \mathbf{E}, \quad VAR \in \mathbf{V} \to \mathbf{E}, \quad LAM \in \mathbf{V} \times \mathbf{E} \to \mathbf{E}, \quad APP \in \mathbf{E} \times \mathbf{E} \to \mathbf{E}$$

Then for any term E with variables from $\mathbf{V}$, we define its representation $\ulcorner E \urcorner \in \mathbf{E}$ in the obvious way, e.g., $\ulcorner \lambda x.E \urcorner = LAM\,(x, \ulcorner E \urcorner)$. Because of the injectivity and disjointness assumptions, for any $e \in \mathbf{E}$, there is *at most one* E such that $\ulcorner E \urcorner = e$; we need not require that all elements of $\mathbf{E}$ represent well-formed lambda-terms, let alone well-typed ones.

(We deliberately use a very concrete representation of terms, rather than a higher-level notion based on abstract syntax with binding constructs such as [12]. Our ultimate goal is to implement the normalization process as a simple functional program, without assuming potentially expensive operations, such as capture-avoiding substitution, as primitives.)

The task is now to construct an interpretation $\mathcal{I}_\mathrm{r}$ of Σ such that we can recover E's normal form from $[\![E]\!]^{\mathcal{I}_\mathrm{r}}$. We want to use the idea from the introduction, but need to account rigorously for "fresh" variable names. Freshness could be captured abstractly in a framework such as Fraenkel-Mostowski sets [14], but this again removes us a level from a direct implementation. Instead, we will explicitly generate non-clashing variable names. Perhaps the simplest scheme for doing so is through de Bruijn levels [2, 11], but we adopt instead a scheme for generating "globally unique", gensym-style names using a monad, as it scales better to the constructions of the next section.

Auxiliary definitions. We first define the name-generation monad $\mathcal{T}_\mathrm{g}$. This is just a state-passing monad atop $\mathcal{T}_\mathrm{l}$; the state is the "next free index":

$$T^\mathrm{g} A = \mathbf{N} \to T^\mathrm{l}(A \times \mathbf{N}) \qquad \eta^\mathrm{g}\, a = \lambda i.\eta^\mathrm{l}\,(a, i) \qquad t \star^\mathrm{g} f = \lambda i.t\,i \star^\mathrm{l} \lambda(a, i').\,f\,a\,i'$$

With respect to T^g, we can define an effectful computation that generates a fresh name, and one that initializes the index within a delimited subcomputation:

$$\begin{aligned} new_\tau &\in T^\mathrm{g}\mathbf{V} & \qquad withct_A &\in T^\mathrm{g} A \to T^\mathrm{l} A \\ new_\tau &= \lambda i.\eta^\mathrm{l}\,(g_i^\tau, i+1) & \qquad withct_A\, t &= t\,0 \star^\mathrm{l} \lambda(a, i').\eta^\mathrm{l}\, a \end{aligned}$$

where the $g_i^\tau \in \mathbf{V}$ are assumed distinct for distinct i. Note that the codomain of $withct_A$ is simply $T^\mathrm{l} A$, i.e., $withct\; t$ represents a side-effect-free, purely functional computation, for any name-generating computation t.

The residualizing interpretation. We can now define a suitable residualizing interpretation $\mathcal{I}_{\mathrm{r}} = (\mathcal{B}_{\mathrm{r}}, \mathcal{C}_{\mathrm{r}})$. For $\mathcal{B}_{\mathrm{r}}$ we take, for all base types b in Σ,

$$\mathcal{B}_{\mathrm{r}}(b) = T^{\mathrm{g}}\mathbf{E}$$

Formalizing the construction from the introduction, we further define, for any Σ-type τ, a pair of functions commonly called *reification* and *reflection*:

$$\begin{aligned}
\downarrow^{\tau} &\in [\![\tau]\!]^{\mathcal{I}_{\mathrm{r}}} \to T^{\mathrm{g}}\mathbf{E} \\
\downarrow^{b} &= \lambda\varepsilon.\,\varepsilon \\
\downarrow^{\tau_1 \to \tau_2} &= \lambda f.\, \mathit{new}_{\tau_1} \star^{\mathrm{g}} \lambda v. \downarrow^{\tau_2} \big(f\,(\uparrow_{\tau_1}(\eta^{\mathrm{g}}\,(\mathit{VAR}\,v)))\big) \star^{\mathrm{g}} \lambda e.\, \eta^{\mathrm{g}}\,(\mathit{LAM}\,(v, e)) \\
\uparrow_{\tau} &\in T^{\mathrm{g}}\mathbf{E} \to [\![\tau]\!]^{\mathcal{I}_{\mathrm{r}}} \\
\uparrow_{b} &= \lambda\varepsilon.\,\varepsilon \\
\uparrow_{\tau_1 \to \tau_2} &= \lambda\varepsilon.\,\lambda a. \uparrow_{\tau_2} \big(\varepsilon \star^{\mathrm{g}} \lambda e. \downarrow^{\tau_1} a \star^{\mathrm{g}} \lambda e'.\, \eta^{\mathrm{g}}\,(\mathit{APP}\,(e, e'))\big)
\end{aligned}$$

(It may be helpful, on a first reading, to think of $\mathcal{T}_{\mathrm{g}}$ as just the identity monad, and new_{τ} as "magically" generating fresh variable names; then the reification function simplifies to precisely the construction of *nf* sketched in Section 1.) Finally, we define the residualizing interpretation of constants by

$$\mathcal{C}_{\mathrm{r}}(c) \in [\![\Sigma(c)]\!]^{\mathcal{I}_{\mathrm{r}}}, \qquad \mathcal{C}_{\mathrm{r}}(c) = \uparrow_{\Sigma(c)} (\eta^{\mathrm{g}}\,(\mathit{CST}\,c))$$

The normalization function. To extract the syntactic normal form from the residualizing meaning of a term, we only need to supply a starting index for name generation. We can thus define an *extraction function*:

$$\mathit{nf}_{\tau} \in [\![\tau]\!]^{\mathcal{I}_{\mathrm{r}}} \to T^{\mathrm{l}}\mathbf{E}, \qquad \mathit{nf}_{\tau} = \lambda a.\, \mathit{withct}_{\mathbf{E}}\,(\downarrow^{\tau} a)$$

Finally, we define the (potentially partial) syntax-to-syntax function *norm* on closed terms $\vdash_{\Sigma} E : \tau$ by

$$\mathit{norm}(E) = \widetilde{E} \quad \text{iff} \quad \mathit{nf}_{\tau}([\![E]\!]^{\mathcal{I}_{\mathrm{r}}}\,\emptyset) = \eta^{\mathrm{l}}\,\ulcorner\widetilde{E}\urcorner$$

(We can find normal forms of open terms by explicitly lambda-abstracting over their free variables. The closed-term formulation leads to a particularly natural implementation, as sketched below.)

Theorem 1 (CBN semantic normalization). *Let $\vdash_{\Sigma} E : \tau$ be a closed Σ-term. Then (0) $\widetilde{E} = \mathit{norm}(E)$ is defined, (1) $\vdash^{\mathrm{nf}} \widetilde{E} : \tau$, (2) $\models \widetilde{E} = E$, and (3) for all $\vdash_{\Sigma} E' : \tau$ such that $\models E' = E$, $\mathit{norm}(E') = \widetilde{E}$.*

If we already know that any Σ-term has a unique (up to α-conversion) $\beta\eta$-long normal form, the proof is fairly simple, using the argument sketched in the introduction. However, it is also possible to prove the theorem directly, using a suitable Kripke logical relation between the meanings of terms in the residualizing interpretation $\mathcal{I}_{\mathrm{r}}$ and in an arbitrary one $\mathcal{I}$, with the base relation taken as the denotational meaning function. The details (for a more general setting, as sketched in Section 5) can be found in [11].

A normalization algorithm. The normalization function described above can be effectively computed as a program in any PCF-like functional language. The key idea is to express the residualizing semantic interpretation as a syntactic *realization* of all base types and constants in $\mathit{\Sigma}$ in terms of types and terms of the programming-language signature $\mathit{\Sigma}_{\mathrm{pl}}$, giving a substitution $\mathit{\Phi}_{\mathrm{r}}$, such that $\llbracket E\{\mathit{\Phi}_{\mathrm{r}}\}\rrbracket^{\mathcal{I}_{\mathrm{pl}}} = \llbracket E\rrbracket^{\mathcal{I}_{\mathrm{r}}}$. Likewise, for any $\mathit{\Sigma}$-type τ, we construct a term nf_τ such that $\llbracket \mathsf{nf}_\tau\rrbracket^{\mathcal{I}_{\mathrm{pl}}}\,\emptyset = \mathit{nf}_\tau$. Then for any closed $\mathit{\Sigma}$-term $\vdash_{\mathit{\Sigma}} E : \tau$, we can compute its normal form by evaluating the $\mathit{\Sigma}_{\mathrm{pl}}$-program "$\mathsf{nf}_\tau\; E\{\mathit{\Phi}_{\mathrm{r}}\}$". Again, the details can be found in [11].

3 Normalization by evaluation for call by value with effects

We now refine the normalization result to a language based on Moggi's computational lambda-calculus λ_c [16], which provides a semantic framework for ML-like languages where "functions" may have effects such as mutating the state, performing input/output operations, or raising exceptions.

3.1 Language and semantic framework

Syntax and semantics. The syntax of types is the same as before. For terms, we also add a let-construct, with the usual typing rule:

$$\frac{\mathit{\Gamma} \vdash_{\mathit{\Sigma}} E_1 : \tau_1 \qquad \mathit{\Gamma}, x{:}\,\tau_1 \vdash_{\mathit{\Sigma}} E_2 : \tau_2}{\mathit{\Gamma} \vdash_{\mathit{\Sigma}} \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 : \tau_2}$$

Now an interpretation $\mathcal{I}$ of a signature $\mathit{\Sigma}$ consists a triple $(\mathcal{B}, \mathcal{T}, \mathcal{C})$. As before, $\mathcal{B}$ assigns cpos to base types of $\mathit{\Sigma}$. The new component $\mathcal{T} = (T, \eta, \star)$ is a monad used to model computational effects. These could be just divergence (modeled with the lifting monad), but also state, exceptions, continuations, etc.; the actual effectful operations are invoked through suitable constants from $\mathit{\Sigma}$.

We need to assume that $\mathcal{T}$ is layered atop $\mathcal{T}_{\mathrm{l}}$ [10]; this amounts to requiring that TA is pointed for any A, and that $\lambda t.\, t \star f \in TA \to TB$ is strict for any $f \in A \to TB$. The CBV semantics of types is then given by:

$$\llbracket b\rrbracket^{\mathcal{I}}_{\mathrm{v}} = \mathcal{B}(b) \qquad\qquad \llbracket \tau_1 \to \tau_2\rrbracket^{\mathcal{I}}_{\mathrm{v}} = \llbracket \tau_1\rrbracket^{\mathcal{I}}_{\mathrm{v}} \to T\llbracket \tau_2\rrbracket^{\mathcal{I}}_{\mathrm{v}}$$

The meaning of a typing environment, $\llbracket \mathit{\Gamma}\rrbracket^{\mathcal{I}}_{\mathrm{v}}$, is a $(\mathrm{dom}\,\mathit{\Gamma})$-indexed product of the meanings of the individual types, as before.

For the semantic function $\mathcal{C}$, we again require that for any $c \in \mathrm{dom}\,\mathit{\Sigma}$, $\mathcal{C}(c) \in \llbracket \mathit{\Sigma}(c)\rrbracket^{\mathcal{I}}_{\mathrm{v}}$. Then we define the meaning of a well-typed term $\mathit{\Gamma} \vdash_{\mathit{\Sigma}} E : \tau$ as a continuous function $\llbracket E\rrbracket^{\mathcal{I}}_{\mathrm{v}} \in \llbracket \mathit{\Gamma}\rrbracket^{\mathcal{I}}_{\mathrm{v}} \to T\llbracket \tau\rrbracket^{\mathcal{I}}_{\mathrm{v}}$, as follows:

$$\begin{array}{ll}
\llbracket c\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \;=\; \eta\,(\mathcal{C}(c)) & \llbracket \lambda x^{\tau}.\, E\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \;=\; \eta\,(\lambda a.\, \llbracket E\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x \mapsto a])) \\
\llbracket x\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \;=\; \eta\,(\rho\, x) & \llbracket E_1\, E_2\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \;=\; \llbracket E_1\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda f.\, \llbracket E_2\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a.\, f\, a
\end{array}$$
$$\llbracket \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \;=\; \llbracket E_1\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a.\, \llbracket E_2\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x \mapsto a])$$

Note that the **let**-construct appears redundant, because $\llbracket \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho = \llbracket (\lambda x.\, E_2)\, E_1\rrbracket^{\mathcal{I}}_{\mathrm{v}}\,\rho$, but including it enables a nicer syntactic characterization of normal forms.

Equivalence and normal forms. Analogously to the CBN case, we write $\models_{\mathrm{v}} E = E'$ if for all $\mathcal{I}$, $[\![E]\!]^{\mathcal{I}}_{\mathrm{v}} = [\![E']\!]^{\mathcal{I}}_{\mathrm{v}}$. The shape of normal forms is now somewhat different, however: instead of normal and atomic forms, we have *normal values* and *normal computations*:

$$\frac{\Sigma(c) = b}{\Gamma \Vdash^{\mathrm{nv}} c : b} \qquad \frac{\Gamma(x) = b}{\Gamma \Vdash^{\mathrm{nv}} x : b} \qquad \frac{\Gamma, x{:}\,\tau_1 \Vdash^{\mathrm{nc}} E : \tau_2}{\Gamma \Vdash^{\mathrm{nv}} \lambda x^{\tau_1}.E : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \Vdash^{\mathrm{nv}} E : \tau}{\Gamma \Vdash^{\mathrm{nc}} E : \tau} \qquad \frac{\Sigma(c) = \tau_1 \to \tau_2 \quad \Gamma \Vdash^{\mathrm{nv}} E : \tau_1 \quad \Gamma, x{:}\,\tau_2 \Vdash^{\mathrm{nc}} E' : \tau}{\Gamma \Vdash^{\mathrm{nc}} \mathbf{let}\ x = c\,E\ \mathbf{in}\ E' : \tau}$$

$$\frac{\Gamma(x') = \tau_1 \to \tau_2 \quad \Gamma \Vdash^{\mathrm{nv}} E : \tau_1 \quad \Gamma, x{:}\,\tau_2 \Vdash^{\mathrm{nc}} E' : \tau}{\Gamma \Vdash^{\mathrm{nc}} \mathbf{let}\ x = x'\,E\ \mathbf{in}\ E' : \tau}$$

That is, a normal value is either a base-typed constant or variable, or of the form $\lambda x.\,\mathbf{let}\ x_1 = f_1\,V_1\ \mathbf{in}\ \cdots\ \mathbf{let}\ x_n = f_n\,V_n\ \mathbf{in}\ V$ where all the Vs are normal values, and each f_i is a function-typed constant or variable.

The set of normal-form terms is similar to Flanagan et al.'s *A-normal forms* [13]. However, their notion of A-reduction does not include even restricted β-conversion, so a term such as $(\lambda x.\,x)\,y$ is already A-normal. (Nor does it include η-like let-conversions: both $f\,x$ and $\mathbf{let}\ y = f\,x\ \mathbf{in}\ y$ are A-normal.) A much closer match is Ohori's language of *cut-free A-normal sequent proofs* [17], but still with one important difference: since we also care about uniqueness of normal forms, a variable is only a normal value in our sense if it is of base type; function-typed normal values must always be syntactic lambda-abstractions.

3.2 A normalization result

Term representations. Corresponding to the extended source syntax, we also assume given an additional constructor function $\mathit{LET} \in \mathbf{V} \times \mathbf{E} \times \mathbf{E} \to \mathbf{E}$, injective and with range disjoint from the others. The representation function for terms is also extended in the obvious way.

Residualizing monad. For constructing the residualizing interpretation, we now also need to pick a residualizing monad $\mathcal{T}_{\mathrm{r}}$. It is easy to see that we cannot simply use the lifting monad here, even if we only care about "purely functional" call-by-value languages, i.e., interpretations with $\mathcal{T}$ taken as lifting. The reason is that the two normal-form terms $E_1 = \lambda f^{b \to b}.\,\lambda x^b.\,\mathbf{let}\ y = f\,x\ \mathbf{in}\ x$ and $E_2 = \lambda f^{b \to b}.\,\lambda x^b.x$ are not semantically equivalent: for any $\mathcal{I}$ where $\mathcal{T}$ is the lifting monad, we only have $[\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}} \sqsubseteq [\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho$ (with the strictness of the inequality demonstrated by application of both sides to $\lambda a.\,\bot$). But $\ulcorner E_1 \urcorner \not\sqsubseteq \ulcorner E_2 \urcorner$, so there can be no *monotone* (let alone continuous) function $\mathit{nf} : [\![(b \to b) \to b \to b]\!]^{\mathcal{I}_{\mathrm{r}}} \to \mathbf{E}_\bot$ such that $\mathit{nf}([\![E_1]\!]^{\mathcal{I}_{\mathrm{r}}}\,\emptyset) = \eta^{\mathrm{l}}\ulcorner E_1 \urcorner$ and $\mathit{nf}([\![E_2]\!]^{\mathcal{I}_{\mathrm{r}}}\,\emptyset) = \eta^{\mathrm{l}}\ulcorner E_2 \urcorner$, if we require $\mathcal{I}_{\mathrm{r}}$ to use only the lifting monad to interpret computational effects.

Indeed, looking at the shape of normal computations, we see that the residualizing monad most allow us to register exactly where and when a function-typed constant or variable was applied, even if its return value is never used. We will show that it suffices that $\mathcal{T}_{\mathrm{r}}$ can be equipped with operations

$$\mathit{bind}_\tau \in \mathbf{E} \to T^{\mathrm{r}}\mathbf{V} \qquad \text{and} \qquad \mathit{collect} \in T^{\mathrm{r}}\mathbf{E} \to T^{\mathrm{g}}\mathbf{E}$$

satisfying the equational constraints

$$\begin{aligned} \mathit{collect}\,(\eta^{\mathrm{r}}\, e) &= \eta^{\mathrm{g}}\, e \\ \mathit{collect}\,(\mathit{bind}_\tau\, e \star^{\mathrm{r}} f) &= \mathit{new}_\tau \star^{\mathrm{g}} \lambda v.\, \mathit{collect}\,(f\, v) \star^{\mathrm{g}} \lambda e'.\, \eta^{\mathrm{g}}\,(\mathit{LET}\,(v,e,e')) \end{aligned}$$

These equations ensure that a $T^{\mathrm{r}}\mathbf{E}$-computation consisting of a sequence of calls to *bind* followed by returning a term has the effect of wrapping that term in a corresponding sequence of *LET*s:

$$\begin{aligned} &\mathit{collect}\,(\mathit{bind}_{\tau_1}\, e_1 \star^{\mathrm{r}} \lambda v_1.\cdots \mathit{bind}_{\tau_n}\,(e_n\, v_1 \cdots v_{n-1}) \star^{\mathrm{r}} \lambda v_n.\, \eta^{\mathrm{r}}\,(e\, v_1 \cdots v_n)) = \\ &\quad \mathit{new}_{\tau_1} \star^{\mathrm{g}} \lambda v_1.\cdots \mathit{new}_{\tau_n} \star^{\mathrm{g}} \lambda v_n.\, \eta^{\mathrm{g}}\,(\mathit{LET}\,(v_1, e_1, \cdots \mathit{LET}\,(v_n, e_n\, v_1 \cdots v_{n-1}, e\, v_1 \cdots v_n))) \end{aligned}$$

To view T^{g}-computations as special cases of T^{r}-computations, we will also need a *monad morphism* from $\mathcal{T}_{\mathrm{g}}$ to $\mathcal{T}_{\mathrm{r}}$, i.e., a collection of functions $\gamma^{\mathrm{g,r}}_A \in T^{\mathrm{g}} A \to T^{\mathrm{r}} A$, such that $\gamma^{\mathrm{g,r}}\,(\eta^{\mathrm{g}}\, a) = \eta^{\mathrm{r}}\, a$ and $\gamma^{\mathrm{g,r}}\,(t \star^{\mathrm{g}} f) = \gamma^{\mathrm{g,r}}\, t \star^{\mathrm{r}} \lambda a.\, \gamma^{\mathrm{g,r}}\,(f\, a)$. We can construct a monad with these operations in several ways, notably including the following two:

The continuation monad with answer domain $T^{\mathrm{g}}\mathbf{E}$. We take:

$$\begin{aligned} T^{\mathrm{r}} A &= (A \to T^{\mathrm{g}}\mathbf{E}) \to T^{\mathrm{g}}\mathbf{E} & \gamma^{\mathrm{g,r}}\, t &= \lambda\kappa.\, t \star^{\mathrm{g}} \kappa \\ \eta^{\mathrm{r}}\, a &= \lambda\kappa.\, \kappa\, a & \mathit{bind}_\tau\, e &= \lambda\kappa.\, \mathit{new}_\tau \star^{\mathrm{g}} \lambda v.\, \kappa\, v \star^{\mathrm{g}} \lambda e'.\, \eta^{\mathrm{g}}\,(\mathit{LET}\,(v,e,e')) \\ t \star^{\mathrm{r}} f &= \lambda\kappa.\, t\,(\lambda a.\, f\, a\, \kappa) & \mathit{collect}\, t &= t\, \eta^{\mathrm{g}} \end{aligned}$$

This continuation-based wrapping of syntactic bindings was originally used for an "administrative-reduction free" continuation-passing transformation [9], and later adapted for a similar purpose in type-directed partial evaluation [7]. It is notable for also allowing an extension of CBV NBE to sum types (see Section 4).

The accumulation monad over the monoid of $(\mathbf{V}\times\mathbf{E})$*-lists.* Writing $[\,]$ for the empty list, $[-]$ for a singleton list, and @ for list concatenation, we take:

$$\begin{aligned} T^{\mathrm{r}} A &= T^{\mathrm{g}}(A \times (\mathbf{V} \times \mathbf{E})^*) & \gamma^{\mathrm{g,r}}\, t &= t \star^{\mathrm{g}} \lambda a.\, \eta^{\mathrm{g}}\,(a, [\,]) \\ \eta^{\mathrm{r}}\, a &= \eta^{\mathrm{g}}\,(a, [\,]) & \mathit{bind}_\tau\, e &= \mathit{new}_\tau \star^{\mathrm{g}} \lambda v.\, \eta^{\mathrm{g}}\,(v, [(v,e)]) \\ t \star^{\mathrm{r}} f &= t \star^{\mathrm{g}} \lambda(a,l).\, f\, a \star^{\mathrm{g}} \lambda(b,l').\, \eta^{\mathrm{g}}\,(b, l\ @\ l') & \mathit{collect}\, t &= t \star^{\mathrm{g}} \lambda(e,l).\, \eta^{\mathrm{g}}\,(\mathit{wrap}\, l\, e) \end{aligned}$$

with the auxiliary function $\mathit{wrap} : (\mathbf{V} \times \mathbf{E})^* \to \mathbf{E} \to \mathbf{E}$ defined inductively as

$$\mathit{wrap}\,[\,]\, e = e \qquad \mathit{wrap}\,([(v,e')]\ @\ l)\, e = \mathit{LET}\,(v, e', \mathit{wrap}\, l\, e)$$

This choice is a refinement of state-based TDPE [18]; see the end of this section for a brief account of the relationship between accumulation and state. Other constructions are also possible, such as accumulation with respect to the monoid $(\mathbf{E} \to \mathbf{E}, \mathit{id}, \circ)$.

Residualizing interpretation. For the residualizing interpretation, we again interpret all base types of Σ as syntactic lambda-terms; this time, however, we do not need to involve the name-generation monad yet, but simply take:

$$\mathcal{B}_{\mathrm{r}}(b) = \mathbf{E}$$

For any definition of $\mathcal{T}_{\mathrm{r}}$ satisfying the equational constraints on *bind* and *collect*, we can then define new reification and reflection functions:

$$
\begin{aligned}
\downarrow^{\tau} &\in [\![\tau]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}} \to T^{\mathrm{g}}\mathbf{E} \\
\downarrow^{b} &= \lambda e.\, \eta^{\mathrm{g}}\, e \\
\downarrow^{\tau_1 \to \tau_2} &= \lambda f.\, \mathit{new}_{\tau_1} \star^{\mathrm{g}} \lambda v.\, \mathit{collect}\, \big(\uparrow_{\tau_1} (\mathit{VAR}\, v) \star^{\mathrm{r}} \lambda a.\, f\, a \star^{\mathrm{r}} \lambda b.\, \gamma^{\mathrm{g,r}} (\downarrow^{\tau_2} b)\big) \star^{\mathrm{g}} \lambda e. \\
&\qquad \eta^{\mathrm{g}} (\mathit{LAM}\, (v, e)) \\[1ex]
\uparrow_{\tau} &\in \mathbf{E} \to T^{\mathrm{r}}[\![\tau]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}} \\
\uparrow_{b} &= \lambda e.\, \eta^{\mathrm{r}}\, e \\
\uparrow_{\tau_1 \to \tau_2} &= \lambda e.\, \eta^{\mathrm{r}}\, \big(\lambda a.\, \gamma^{\mathrm{g,r}} (\downarrow^{\tau_1} a) \star^{\mathrm{r}} \lambda e'.\, \mathit{bind}_{\tau_2} (\mathit{APP}\, (e, e')) \star^{\mathrm{r}} \lambda v.\, \uparrow_{\tau_2} (\mathit{VAR}\, v)\big)
\end{aligned}
$$

(The codomain of $\uparrow_{\tau}$ is $T^{\mathrm{r}}[\![\tau]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}}$, rather than simply $[\![\tau]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}}$, to accommodate the extensions in Section 4.) Note in particular how every construction of an *APP*-term is wrapped in a *bind*.

Finally, as for call by name, we interpret all Σ-constants as reflected *CST*-constructors:

$$
\mathcal{C}_{\mathrm{r}}(c) \in [\![\Sigma(c)]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}}, \qquad \mathcal{C}_{\mathrm{r}}(c) = a, \text{ where } \uparrow_{\Sigma(c)} (\mathit{CST}\, c) = \eta^{\mathrm{r}}\, a
$$

($\mathcal{C}_{\mathrm{r}}(c)$ is well defined, because the reflection function factors through the injective η^{r}.) We also define the extraction function essentially as before:

$$
\mathit{nf}_{\tau} \in [\![\tau]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}} \to T^{\mathrm{l}}\mathbf{E}, \qquad \mathit{nf}_{\tau} = \lambda a.\, \mathit{withct}_{\mathbf{E}} (\downarrow^{\tau} a)
$$

and the CBV normalization function for a closed value (constant or lambda-abstraction) E as

$$
\mathit{norm}_{\mathrm{v}}(E) = \widetilde{E} \quad \text{iff} \quad \mathit{nf}_{\tau}\, a = \eta^{\mathrm{l}}\, \ulcorner \widetilde{E} \urcorner, \text{ where } [\![E]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}}\, \emptyset = \eta^{\mathrm{r}}\, a.
$$

(We can find the normal form of a non-value term by wrapping a dummy lambda-abstraction around it.)

Theorem 2 (CBV semantic normalization). *Let* $\vdash_{\Sigma} E : \tau$ *be a value, and take* $\widetilde{E} = \mathit{norm}_{\mathrm{v}}(E)$. *Then (0)* $\widetilde{E}$ *is defined, (1)* $\vdash^{\mathrm{nv}} \widetilde{E} : \tau$, *(2)* $\models_{\mathrm{v}} \widetilde{E} = E$, *and (3) if* $\models_{\mathrm{v}} E' = E$ *then* $\mathit{norm}_{\mathrm{v}}(E') = \widetilde{E}$.

The proof is similar to the CBN case, but using a pair of mutually inductively defined logical relations, one for values and one for computations. Very roughly, one again establishes that the residualizing and the arbitrary interpretations of all terms are related, and that for a pair of related values $(a, a') \in [\![\tau]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{r}}} \times [\![\tau]\!]_{\mathrm{v}}^{\mathcal{I}}$ the $\mathcal{I}$-meaning of $\downarrow^{\tau} a$ equals a'.

A normalization algorithm. Phrasing the normalization function as a functional program is a bit more complicated than for the CBN case. A typical CBV host language will have its own notion of effects, modeled by some monad $\mathcal{T}_{\mathrm{pl}}$, which is not likely to be exactly our residualizing monad $\mathcal{T}_{\mathrm{r}}$. However, much as we can embed the normalization algorithm into a host language with a signature much larger than what we need for the construction, we can realize both the residualizing and the name-generating monad through a uniform *effect-embedding* into a more general notion of effect in the host language [10].

For example, the accumulation-based choice of $\mathcal{T}_{\mathrm{r}}$ can be easily (and more efficiently) implemented by passing around the bindings accumulated so far in a mutable state cell, rather than appending the bindings from both subcomputations in $\star$. The current name-generation index is naturally kept in another cell. An analogous, but somewhat more involved, construction also allows us to simulate $\mathcal{T}_{\mathrm{r}}$ taken as a continuation monad, provided the host language provides both (higher-typed) state and first-class continuations, as found in Scheme or SML/NJ. The resulting implementation forms the basis of the CBV normalization algorithm used in the context of type-directed partial evaluation [7].

4 Structured data types

In this section, we consider normalization for the call-by-value language extended with product and sum types. (Adding products to the call-by-name language is trivial, but it does not appear possible to add even weak sum types, at least in the domain-theoretic semantics.)

Syntax. We extend the set of types by two new type constructors:

$$\tau ::= \cdots \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2$$

(The generalizations to n-ary ($n \geq 0$) products and sums are completely straightforward and thus omitted.) The associated new terms are:

$$\frac{\Gamma \vdash_\Sigma E_1 : \tau_1 \quad \Gamma \vdash_\Sigma E_2 : \tau_2}{\Gamma \vdash_\Sigma (E_1, E_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash_\Sigma E : \tau_1 \times \tau_2 \quad \Gamma, x_1{:}\tau_1, x_2{:}\tau_2 \vdash_\Sigma E' : \tau}{\Gamma \vdash_\Sigma \mathbf{split}\,(E, x_1.\,x_2.\,E') : \tau}$$

$$\frac{\Gamma \vdash_\Sigma E : \tau_1}{\Gamma \vdash_\Sigma \mathbf{inl}\,(E) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash_\Sigma E : \tau_2}{\Gamma \vdash_\Sigma \mathbf{inr}\,(E) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash_\Sigma E : \tau_1 + \tau_2 \quad \Gamma, x_1{:}\tau_1 \vdash_\Sigma E_1 : \tau \quad \Gamma, x_2{:}\tau_2 \vdash_\Sigma E_2 : \tau}{\Gamma \vdash_\Sigma \mathbf{case}\,(E, x_1.\,E_1, x_2.\,E_2) : \tau}$$

(Instead of **split**, we could have used explicit projections, but the characterization of normal forms becomes more uniform with **split**. In practice, the separate split-construct above is usually folded into pattern-matching let- and lambda-bindings.)

Semantics. The semantics of the type constructors is standard:

$$\begin{aligned}
[\![\tau_1 \times \tau_2]\!]^{\mathcal{I}}_{\mathrm{v}} &= [\![\tau_1]\!]^{\mathcal{I}}_{\mathrm{v}} \times [\![\tau_2]\!]^{\mathcal{I}}_{\mathrm{v}} = \{(a_1, a_2) \mid a_1 \in [\![\tau_1]\!]^{\mathcal{I}}_{\mathrm{v}}, a_2 \in [\![\tau_1]\!]^{\mathcal{I}}_{\mathrm{v}}\} \\
[\![\tau_1 + \tau_2]\!]^{\mathcal{I}}_{\mathrm{v}} &= [\![\tau_1]\!]^{\mathcal{I}}_{\mathrm{v}} + [\![\tau_2]\!]^{\mathcal{I}}_{\mathrm{v}} = \{\iota_1\, a \mid a \in [\![\tau_1]\!]^{\mathcal{I}}_{\mathrm{v}}\} \cup \{\iota_2\, a \mid a \in [\![\tau_2]\!]^{\mathcal{I}}_{\mathrm{v}}\}
\end{aligned}$$

as is the semantics of the associated terms:

$$\begin{aligned}
[\![(E_1, E_2)]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &= [\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a_1.\,[\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a_2.\,\eta\,(a_1, a_2) \\
[\![\mathbf{split}\,(E, x_1.\,x_2.\,E')]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &= [\![E]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda(a_1, a_2).\,[\![E']\!]^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x_1 \mapsto a_1, x_2 \mapsto a_2])
\end{aligned}$$

$$
\begin{aligned}
[\![\mathbf{inl}\,(E)]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &= [\![E]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a.\,\eta\,(\iota_1\,a)\\
[\![\mathbf{inr}\,(E)]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &= [\![E]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a.\,\eta\,(\iota_2\,a)\\
[\![\mathbf{case}\,(E, x_1.\,E_1, x_2.\,E_2)]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &= [\![E]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda s.\begin{cases} [\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x_1 \mapsto a_1]) & \text{if } s = \iota_1\,a_1\\ [\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x_2 \mapsto a_2]) & \text{if } s = \iota_2\,a_2\end{cases}
\end{aligned}
$$

Normal forms. With the addition of product and sum types, CBV normal forms exhibit a striking similarity with cut-free proofs in Gentzen-style intuitionistic sequent calculus, as also noted by Ohori [17]. In fact, we also get the usual sequent-calculus inconvenience of having to make arbitrary choices about the order in which we apply left-rules to decompose the types of variables. To keep normal forms unique, we choose to eliminate structured-type variables immediately as they are introduced, in a stack-like manner. (Immediate elimination leads to a slight anomaly for constants introduced by the signature: we can only allow Σ to declare constants of base and top-level-functional types. In the rare cases where we need, e.g., a sum-typed constant $c : \tau_1 + \tau_2$ in Σ, it can be provided as a function $c' : 1 \to \tau_1 + \tau_2$ where 1 is the zero-ary product type.)

We now have three mutually recursive notions of normality: normal values $\Gamma \vdash^{\mathrm{nv}} E : \tau$, normal computations $\Gamma \vdash^{\mathrm{nc}} E : \tau$, and normal bodies $\Gamma \mid \Theta \vdash^{\mathrm{nb}} E : \tau$, where Θ is an *ordered* list of typing assumptions:

$$
\frac{\Sigma(c) = b}{\Gamma \vdash^{\mathrm{nv}} c : b}
\qquad
\frac{\Gamma(x) = b}{\Gamma \vdash^{\mathrm{nv}} x : b}
\qquad
\frac{\Gamma \mid x{:}\,\tau_1 \vdash^{\mathrm{nb}} E : \tau_2}{\Gamma \vdash^{\mathrm{nv}} \lambda x^{\tau_1}.\,E : \tau_1 \to \tau_2}
$$

$$
\frac{\Gamma \vdash^{\mathrm{nv}} E_1 : \tau_1 \quad \Gamma \vdash^{\mathrm{nv}} E_2 : \tau_2}{\Gamma \vdash^{\mathrm{nv}} (E_1, E_2) : \tau_1 \times \tau_2}
\qquad
\frac{\Gamma \vdash^{\mathrm{nv}} E : \tau_1}{\Gamma \vdash^{\mathrm{nv}} \mathbf{inl}\,(E) : \tau_1 + \tau_2}
\qquad
\frac{\Gamma \vdash^{\mathrm{nv}} E : \tau_2}{\Gamma \vdash^{\mathrm{nv}} \mathbf{inr}\,(E) : \tau_1 + \tau_2}
$$

$$
\frac{\Gamma \vdash^{\mathrm{nv}} E : \tau}{\Gamma \vdash^{\mathrm{nc}} E : \tau}
\qquad
\frac{\Sigma(c) = \tau_1 \to \tau_2 \quad \Gamma \vdash^{\mathrm{nv}} E : \tau_1 \quad \Gamma \mid x{:}\,\tau_2 \vdash^{\mathrm{nb}} E' : \tau}{\Gamma \vdash^{\mathrm{nc}} \mathbf{let}\ x = c\,E\ \mathbf{in}\ E' : \tau}
$$

$$
\frac{\Gamma(x') = \tau_1 \to \tau_2 \quad \Gamma \vdash^{\mathrm{nv}} E : \tau_1 \quad \Gamma \mid x{:}\,\tau_2 \vdash^{\mathrm{nb}} E' : \tau}{\Gamma \vdash^{\mathrm{nc}} \mathbf{let}\ x = x'\,E\ \mathbf{in}\ E' : \tau}
$$

$$
\frac{\Gamma \vdash^{\mathrm{nc}} E : \tau}{\Gamma \mid\ \vdash^{\mathrm{nb}} E : \tau}
\qquad
\frac{\Gamma, x{:}\,b \mid \Theta \vdash^{\mathrm{nb}} E : \tau}{\Gamma \mid \Theta, x{:}\,b \vdash^{\mathrm{nb}} E : \tau}
\qquad
\frac{\Gamma, x{:}\,\tau_1 \to \tau_2 \mid \Theta \vdash^{\mathrm{nb}} E : \tau}{\Gamma \mid \Theta, x{:}\,\tau_1 \to \tau_2 \vdash^{\mathrm{nb}} E : \tau}
$$

$$
\frac{\Gamma \mid \Theta, x_1{:}\,\tau_1, x_2{:}\,\tau_2 \vdash^{\mathrm{nb}} E : \tau}{\Gamma \mid \Theta, x{:}\,\tau_1{\times}\tau_2 \vdash^{\mathrm{nb}} \mathbf{split}\,(x, x_1.\,x_2.\,E) : \tau}
\qquad
\frac{\Gamma \mid \Theta, x_1{:}\,\tau_1 \vdash^{\mathrm{nb}} E_1 : \tau \quad \Gamma \mid \Theta, x_2{:}\,\tau_2 \vdash^{\mathrm{nb}} E_2 : \tau}{\Gamma \mid \Theta, x{:}\,\tau_1{+}\tau_2 \vdash^{\mathrm{nb}} \mathbf{case}\,(x, x_1.\,E_1, x_2.\,E_2) : \tau}
$$

Note how newly-introduced variables are put into the quarantined context Θ, where their types are decomposed and the pieces migrate back into the ordinary context Γ. In particular, without the rules for product and sum types, the new definitions of normal values and computations agree exactly with the original ones.

Normalization by evaluation. We show only how the continuation-based residualizing interpretation can be extended. Products could be added to an accumulation-based interpretation without too much trouble, but sums apparently require the full power of applying a single continuation multiple times.

We assume the syntax-constructor functions for **E** are extended with functions *PAIR*, *SPLIT*, *INL*, *INR*, and *CASE* with the obvious types. Then we can define additional helper functions analogous to bind_τ from before:

$$\begin{aligned}
&\mathit{bindp}_{\tau_1,\tau_2} \in \mathbf{E} \to T^{\mathrm{r}}(\mathbf{V} \times \mathbf{V}) \\
&\quad = \lambda e.\, \lambda \kappa.\, \mathit{new}_{\tau_1} \star^{\mathrm{g}} \lambda v_1.\, \mathit{new}_{\tau_2} \star^{\mathrm{g}} \lambda v_2.\, \kappa\,(v_1, v_2) \star^{\mathrm{g}} \lambda e'.\, \eta^{\mathrm{g}}\,(\mathit{SPLIT}\,(e, v_1, v_2, e')) \\
&\mathit{binds}_{\tau_1,\tau_2} \in \mathbf{E} \to T^{\mathrm{r}}(\mathbf{V} + \mathbf{V}) \\
&\quad = \lambda e.\, \lambda \kappa.\, \mathit{new}_{\tau_1} \star^{\mathrm{g}} \lambda v_1.\, \mathit{new}_{\tau_2} \star^{\mathrm{g}} \lambda v_2.\, \kappa\,(\iota_1\, v_1) \star^{\mathrm{g}} \lambda e_1.\, \kappa\,(\iota_2\, v_2) \star^{\mathrm{g}} \lambda e_2. \\
&\qquad\qquad \eta^{\mathrm{g}}\,(\mathit{CASE}\,(e, v_1, e_1, v_2, e_2))
\end{aligned}$$

and the corresponding cases for the reification and reflection functions, extending the ones from Section 2.2:

$$\begin{aligned}
\downarrow^{\tau_1 \times \tau_2} &= \lambda(a_1, a_2).\, \downarrow^{\tau_1} a_1 \star^{\mathrm{g}} \lambda e_1.\, \downarrow^{\tau_2} a_2 \star^{\mathrm{g}} \lambda e_2.\, \eta^{\mathrm{g}}\,(\mathit{PAIR}\,(e_1, e_2)) \\
\downarrow^{\tau_1 + \tau_2} &= \lambda s.\, \begin{cases} \downarrow^{\tau_1} a_1 \star^{\mathrm{g}} \lambda e_1.\, \eta^{\mathrm{g}}\,(\mathit{INL}\, e_1) & \text{if } s = \iota_1\, a_1 \\ \downarrow^{\tau_2} a_2 \star^{\mathrm{g}} \lambda e_2.\, \eta^{\mathrm{g}}\,(\mathit{INR}\, e_2) & \text{if } s = \iota_2\, a_2 \end{cases}
\end{aligned}$$

$$\begin{aligned}
\uparrow_{\tau_1 \times \tau_2} &= \lambda e.\, \mathit{bindp}_{\tau_1,\tau_2}\, e \star^{\mathrm{r}} \lambda(v_1, v_2).\, \uparrow_{\tau_1} (\mathit{VAR}\, v_1) \star^{\mathrm{r}} \lambda a_1.\, \uparrow_{\tau_2} (\mathit{VAR}\, v_2) \star^{\mathrm{r}} \lambda a_2.\, \eta^{\mathrm{r}}\,(a_1, a_2) \\
\uparrow_{\tau_1 + \tau_2} &= \lambda e.\, \mathit{binds}_{\tau_1,\tau_2}\, e \star^{\mathrm{r}} \lambda s.\, \begin{cases} \uparrow_{\tau_1} (\mathit{VAR}\, v_1) & \text{if } s = \iota_1\, v_1 \\ \uparrow_{\tau_2} (\mathit{VAR}\, v_2) & \text{if } s = \iota_2\, v_2 \end{cases}
\end{aligned}$$

The residualizing interpretation is as before. (Note that we cannot give a residualizing interpretation to constants of top-level product or sum type, since the reflection function does *not* factor through η^{r} in these cases.) The normalization function and Theorem 2 remain the same. For the implementation, the realizations of *bindp*, *binds*, and the new clauses for reification and reflection in terms of continuation-manipulating primitives are straightforward.

5 Type-directed partial evaluation

A primary application of normalization by evaluation is for type-directed partial evaluation (TDPE) [7]. The goal is to simplify a partially applied function of multiple arguments by propagating the values of the known arguments throughout the body of the function. Here, in addition to eliminating β-redexes, we also want to simplify occurrences of constants, such as arithmetic operations, when they are applied to literal values. In other words, we now want to normalize terms with *interpreted* base types and constants. There are in fact two natural ways to achieve this, both expressible in terms of the notion of constrained interpretations:

Offline TDPE. For offline TDPE [7, Section 3], we say that a program is *binding-time separated* if it is expressed over a signature Σ partitioned into a static and a dynamic part, Σ_{s} and Σ_{d}, each containing some type and term constants. The interpretation is likewise partitioned into $\mathcal{I}_{\mathrm{s}}$ and $\mathcal{I}_{\mathrm{d}}$. We then constrain the allowable interpretations so that $\mathcal{I}_{\mathrm{s}}$ is always the standard interpretation (i.e., int as $\mathbf{Z}$, $+$ as addition, fix as the domain-theoretic least fixed point, etc.), while the dynamic part remains completely unconstrained. That is, we consider the notion of *static equivalence*, $\models^{\mathit{Int}_{\mathrm{s}}} E = E'$ where $\mathit{Int}_{\mathrm{s}} = \{\mathcal{I} \mid \mathcal{I}|_{\Sigma_{\mathrm{s}}} = \mathcal{I}_{\mathrm{s}}\}$. When the static normal form $\widetilde{E}$ of a term E exists, it is then equivalent to E with respect to all interpretations of Σ_{d}. The residualizing interpretation of Σ, like any interpretation in $\mathit{Int}_{\mathrm{s}}$, also uses the standard interpretation of Σ_{s}, and the syntax-reconstructing interpretation from Section 3.2 for $\mathcal{I}_{\mathrm{d}}$.

The separation allows us to realize the static part of the signature completely natively in terms of the corresponding construct of the programming language, and in fact we can use syntactic conveniences of the host language, such as pattern matching or **letrec**-forms directly for the static computations. It also becomes possible to self-apply the partial evaluator (the so-called second Futamura projection) [15].

Online TDPE. In the online variant [7, Section 4], like in general online partial evaluation, we do not annotate types, nor most occurrences of constants, with their binding times. Instead, the partial evaluator "opportunistically" propagates statically known data and performs reductions such as $2 + 3 \to 5$.

This corresponds to normalizing with respect to the set $\mathit{Int}_{\mathrm{c}}$ of interpretations $(\mathcal{B}, \mathcal{C})$ that satisfy constraints such as $\mathcal{C}(+)\,(\mathcal{C}(2), \mathcal{C}(3)) = \mathcal{C}(5)$, and possibly also additional ones, such as $\mathcal{C}(+)\,(x, \mathcal{C}(0)) = x$ for all x in $\mathcal{B}(\mathsf{int})$. Again, the standard interpretation of $\mathcal{C}(+)$ satisfies these constraints automatically. The residualizing one includes explicit checks for the reducible cases, to avoid constructing the corresponding redexes in the generated code. This formulation is similar to recent work on merging the reduction-free normalization of function abstraction and application with explicit reduction rules for constants [3].

The advantage of the online approach is that we do not have to explicitly separate the binding times in the source program, but correspondingly it becomes less predictable how much of the source program can be simplified at partial-evaluation time. An online normalizer also requires all primitive operations to explicitly check whether their arguments are literals (so the operation can be eliminated) or more general expressions (so the operation must remain in the normal form), slowing down the specialization process somewhat. Finally, fixed-point operators must still either be explicitly classified as static or dynamic, or need some ad hoc mechanism for deciding whether their unfolding equation should be applied.

In both cases, the normalization function may be partial, i.e., without part (0) of Theorems 1 and 2. This is unavoidable, since in the presence of recursion, some terms simply have no normal form. However, for the CBN case, one can still show that when it is defined, $\widetilde{E}$ satisfies parts (1-3), and also that whenever a $\widetilde{E}$ satisfying (1-3) exists, $\mathit{norm}(E)$ is defined [11]. The situation for CBV TDPE has not been fully analyzed yet, although it seems reasonable to conjecture an analogous result.

In any case, the semantic treatment of TDPE allows us to uniformly analyze the construction and state its correctness criterion independently of the details of its implementation. That is, we can think about partial evaluation in terms of normalization with respect to a class of interpretations, without worrying about whether the normalization is achieved through repeated reductions, or through reduction-less normalization by evaluation.

6 Conclusions and future work

We have seen that the same basic idea that allows us to compute normal forms of lambda-terms with respect to purely functional interpretations also allows us to compute such normal forms with respect to general computational interpretations. In both cases, we chose a "quasi-syntactic" interpretation of the types and constants, and in the latter, also a binding-accumulating monad as the interpretation of computational effects. Both variants of semantic normalization can be phrased as functional-program evaluation, although the construction is significantly more involved in the computational case.

An important application of the normalization construction is type-directed partial evaluation, which seeks to compute normal forms not with respect to *all* interpretations of a signature, but only with respect to a subset of those; different choices of such subsets lead to offline and online partial evaluation. The offline, call-by-name case is analyzed in isolation in an earlier paper [11], but the more general constrained-interpretation formulation presented here seems worth investigating further, especially since it also leads to natural TDPE formulations of other partial-evaluation concepts, such as polyvariant program-point specialization expressed as suitable constraints on the recursion operator.

Additionally, it should be possible to obtain a syntactic analog of the CBV normalization result, showing that the computed normal form is not only equivalent to the original term with respect to arbitrary set-, or domain-theoretic interpretations, but is provably equal to it using the axioms of the computational lambda-calculus [16]. However, it seems that the semantic characterization may be ultimately more convenient for reasoning about programs, since it seems to scale more directly to partially constrained interpretations, and especially static recursion.

Acknowledgments

The author wishes to thank Olivier Danvy and Peter Dybjer for many fruitful discussions of the topics presented here, and the anonymous TLCA'01 reviewers for numerous helpful suggestions (of which regrettably not all could be followed here, due to space constraints.)

References

1. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science, 6th International Conference*, number 953 in Lecture Notes in Computer Science, 1995.
2. 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.

3. Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In *Prospects for Hardware Foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, 1998.
4. 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.
5. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
6. Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.
7. Olivier Danvy. Type-directed partial evaluation. In J. Hatcliff, T. Æ. Mogensen, and P. Thieman, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411. Springer-Verlag, Copenhagen, Denmark, July 1998.
8. Olivier Danvy and Peter Dybjer, editors. *Preliminary Proceedings of the APPSEM Workshop on Normalization by Evaluation.* Department of Computer Science, University of Aarhus, May 1998. BRICS Note NS-98-1.
9. Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
10. Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999.
11. Andrzej Filinski. A semantic account of type-directed partial evaluation. In G. Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999.
12. Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 193–202, Trento, Italy, July 1999.
13. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.
14. Murdoch Gabbay and Andrew Pitts. A new approach to abstract syntax involving binders. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 214–224, Trento, Italy, July 1999.
15. Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32. ACM Press, January 2000.
16. 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.
17. Atsushi Ohori. A Curry-Howard isomorphism for compilation and program execution. In J.-Y. Girard, editor, *Typed Lambda-Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, L'Aquila, Italy, April 1999.
18. Eijiro Sumii and Naoki Kobayashi. Online-and-offline partial evaluation: A mixed approach. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–21. ACM Press, January 2000.