Normalization and Partial Evaluation

Peter Dybjer[1] and Andrzej Filinski[2]*

[1] Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden. Email: peterd@cs.chalmers.se.
[2] Department of Computer Science, University of Copenhagen, Denmark. Email: andrzej@diku.dk

Abstract. We give an introduction to normalization by evaluation and type-directed partial evaluation. We first present normalization by evaluation for a combinatory version of Gödel System T. Then we show normalization by evaluation for typed lambda calculus with β and η conversion. Finally, we introduce the notion of binding time, and explain the method of type-directed partial evaluation for a small PCF-style functional programming language. We give algorithms for both call-by-name and call-by-value versions of this language.

1 Introduction

Normalization. By "normalization" we mean the process known from proof theory and lambda calculus of simplifying proofs or terms in logical systems. Normalization is typically specified as a stepwise simplification process. Formally, one introduces a relation $\mathtt{red}_1$ of step-wise reduction: $E \ \mathtt{red}_1\ E'$ means that E reduces to E' in one step, where E and E' are terms or proof trees.

A "normalization proof" is a proof that a term E can be step-wise reduced to a normal form E' where no further reductions are possible. One distinguishes between "weak" normalization, where one only requires that there exists a reduction to normal form, and "strong" normalization where all reduction sequences must terminate with a normal form.

Partial evaluation. By "partial evaluation" we refer to the process known from computer science of simplifying a program where some of the inputs are known (or "static"). The simplified (or "residual") program is typically obtained by executing operations which only depend on known inputs. More precisely, given a program $\vdash P : \tau_s \times \tau_d \to \tau_r$ of two arguments, and a fixed static argument $s : \tau_s$, we wish to produce a *specialized* program $\vdash P_s : \tau_d \to \tau_r$ such that for all remaining "dynamic" arguments $d : \tau_d$, $\mathrm{eval}\big(P_s\, d\big) = \mathrm{eval}\big(P\,(s,d)\big)$. Hence, running the specialized program P_s on an arbitrary dynamic argument is equivalent to running the original program on both the static and the dynamic ones. In general we may have several static and several dynamic arguments. Writing

* Part of this work was carried out at BRICS (Basic Research in Computer Science, www.brics.dk), funded by the Danish National Research Foundation.

a partial evaluator is therefore like proving an S^m_n-theorem for the programming language: given a program with $m+n$ inputs, m of which are static (given in advance) and n are dynamic, the partial evaluator constructs another program with n inputs which computes the same function of these dynamic inputs as the original one does (with the static inputs fixed). Of course, the goal is not to construct *any* such program but an efficient one!

In a functional language, it is easy to come up with a specialized program P_s: just take $P_s = \lambda d. P(s,d)$. That is, we simply invoke the original program with a constant first argument. But this 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:

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

Suppose we want to compute the third power of several numbers. We can achieve this by 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

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

Using further arithmetic identities, we can 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 should diverge for negative values of n.)

Partial evaluation as normalization. Clearly partial evaluation and normalization are related processes. In partial evaluation, one tries to simplify the original program by executing those operations that only depend on the static inputs. Such a simplification is similar to what happens in normalization. In a functional language, in particular, we can view specialization 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. For example, power_3 is transformed into power'_3 by normalizing power_3 according to the reduction rules mentioned above.

In these lecture notes, we address the question of whether one can apply methods developed for theoretical purposes in proof theory and lambda calculus to achieve partial evaluation of programs. Specifically, we show how the relatively recent idea of *normalization by evaluation*, originally developed by proof-theorists, provides a new approach to the partial evaluation of functional programs, yielding *type-directed partial evaluation.*

Evaluation. By "evaluation" we mean the process of computing the output of a program when given all its inputs. In lambda calculus "evaluation" means normalization of a closed term, usually of base type.

It is important here to contrast (a) *normalization* and *partial evaluation* and (b) the *evaluation* of a complete program. In (a) there are still unknown (dynamic) inputs, whereas in (b) all the inputs are known.

Normalization by evaluation (NBE). Normalization by evaluation is based on the idea that one can obtain a normal form by first interpreting the term in a suitable model, possibly a non-standard one, and then write a function "reify" which maps an object in this model to a normal form representing it. The normalization function is obtained by composing reify with the non-standard interpretation function $[\![-]\!]$:

$$\text{norm } E = \text{reify } [\![E]\!]$$

We want a term to be convertible (provably equal, in some formal system) to its normal form

$$\vdash E = \text{norm } E$$

and therefore we require that reify is a left inverse of $[\![-]\!]$. The other key property is that the interpretation function should map convertible terms to equal objects in the model

$$\text{if } \vdash E = E' \text{ then } [\![E]\!] = [\![E']\!]$$

because then the normalization function maps convertible terms to syntactically equal terms

$$\text{if } \vdash E = E' \text{ then norm } E = \text{norm } E'$$

It follows that the normalization function picks a representative from each equivalence class of convertible terms

$$\vdash E = E' \text{ iff norm } E = \text{norm } E'$$

If the norm-function is computable, we can thus decide whether two terms are convertible by computing their normal forms and comparing them.

This approach bypasses the traditional notion of reduction, formalized as a binary relation, and is therefore sometimes referred to as "reduction-free normalization".

Normalization by evaluation was invented by Martin-Löf [ML75b]. In the original presentation it just appears as a special way of presenting an ordinary normalization proof. Instead of proving that every term has a normal form, one writes a function which returns the normal form, together with a proof that it actually is a normal form. This way of writing a normalization proof is particularly natural from a constructive point of view: to prove that there *exists* a normal form of an arbitrary term, means to actually be able to compute this normal form from the term.

Martin-Löf viewed this kind of normalization proof as normalization by intuitionistic model construction: he pointed out that equality (convertibility) in the object-language is modelled by "definitional equality" in the meta-language [ML75a]. Thus the method of normalization works because the simplification according to this definitional equality is carried out by the evaluator of the intuitionistic (!) meta-language: hence "normalization by evaluation". If instead we work in a classical meta-language, then some extra work would be needed to implement the meta-language function in a programming language.

Martin-Löf's early work on NBE dealt with normalization for intuitionistic type theory [ML75b]. This version of type theory had a weak notion of reduction, where no reduction under the λ-sign was allowed. This kind of reduction is closely related to reduction of terms in combinatory logic, and we shall present this case in Section 2.

Normalization by evaluation for typed lambda calculus with β and η conversion was invented by Berger and Schwichtenberg [BS91]. Initially, they needed a normalization algorithm for their proof system MINLOG, and normalization by evaluation provided a simple solution. Berger then noticed that the NBE program could be extracted from a normalization proof using Tait's method [Ber93]. The method has been refined using categorical methods [AHS95,ČDS98], and also extended to System F [AHS96] and to include strong sums [ADHS01].

Type-directed partial evaluation (TDPE). Type-directed partial evaluation stems from the study of "two-level η-expansions", a technique for making programs specialize better [DMP95]. This technique had already been put to use to write a "one-pass" CPS transformation [DF90] and it turned out to be one of the paths leading to the discovery of NBE [DD98].

Because of its utilization of a standard evaluator for a functional language to achieve normalization, TDPE at first also appeared as a radical departure from traditional, "syntax-directed" partial-evaluation techniques. It only gradually became apparent that the core of TDPE could in fact be seen as a generalization of earlier work on λ-MIX [Gom91], a prototypical partial evaluator for lambda-terms. In particular, the semantic justifications for the two algorithms are structurally very similar. In these notes we present such a semantic reconstruction of the TDPE algorithm as an instance of NBE, based on work by Filinski [Fil99b,Fil01].

Meta-languages for normalization by evaluation and type-directed partial evaluation. NBE is based on the idea that normalization is achieved by interpreting an object-language term as a meta language term and then evaluating the latter. For this purpose one can use different meta languages, and we make use of several such in these notes.

In Section 2 we follow Martin-Löf [ML75b] and Coquand and Dybjer [CD93b] who used an intuitionistic meta-language, which is both a mathematical language and a programming language. Martin-Löf worked in an informal intuitionistic meta-language, where one uses that a function is a function in the intuitionistic sense, that is, an algorithm. Coquand and Dybjer [CD93b] implemented a formal construction similar to Martin-Löf's in the meta-language of Martin-Löf's intuitionistic type theory using the proof assistant ALF [MN93]. This NBE-algorithm makes essential use of the dependent type structure of that language. It is important to point out that the development in Section 2 can be directly understood as a mathematical description of a normalization function also from a classical, set-theoretic point of view. The reason is that Martin-Löf type theory has a direct set-theoretic semantics, where a function space is interpreted classically as the set of all functions in the set-theoretic sense. However, if read with classical eyes, nothing guarantees a priori that the constructed normalization function is computable, so some further reasoning to prove this property would be needed.

We also show how to program NBE-algorithms using a more standard functional language such as Standard ML. Without dependent types we collect all object-language terms into one type and all semantic objects needed for the interpretation into another. With dependent types we can index both the syntactic sets of terms $\mathrm{T}(\tau)$ and the semantic sets $[\![\tau]\!]$ by object-language types τ.

Another possibility, not pursued here, is to use an untyped functional language as a meta-language. For example, Berger and Schwichtenberg's first implementation of NBE in the MINLOG system was written in Scheme.

In type-directed partial evaluation one wishes to write NBE-functions which do not necessarily terminate. Therefore one cannot use Martin-Löf type theory, since it only permits terminating functions. Nevertheless, the dependent type structure would be useful here too, and one might want to use a version of dependent type theory, such as the programming language Cayenne [Aug98], which allows non-terminating functions. We do not pursue this idea further here either.

Notation. NBE-algorithms use the interpretation of an object language in a meta-language. Since our object- and meta-languages are similar (combinatory and lambda calculi), we choose notations which clearly distinguish object- and meta-level but at the same time enhance their correspondence. To this end we use the following conventions.

- Alphabetic object-language constants are written in $\mathsf{sans\ serif}$ and meta-language constants in roman. For example, SUCC denotes the syntactic successor combinator in the language of Section 2 and succ denotes the semantic successor function in the meta-language.
- For symbolic constants, we put a dot above a meta-language symbol to get the corresponding object-language symbol. For example, $\dot{\rightarrow}$ is object-language function space whereas $\rightarrow$ is meta-language function space; $\dot{\lambda}$ is lambda abstraction in the object-language whereas λ is lambda abstraction in the meta-language.
- We also use some special notations. For example, syntactic application is denoted by a dot $(F{\cdot}E)$ whereas semantic application is denoted by juxtaposition $(f\ e)$ as usual. Syntactic pairing uses round brackets (E, E') whereas semantic pairing uses angular ones $\langle e, e'\rangle$.

Plan. The remainder of the chapter is organized as follows.

In Section 2 we introduce a combinatory version of Gödel System T, that is, typed combinatory logic with natural numbers and primitive recursive functionals. We show how to write an NBE-algorithm for this language by interpreting it in a non-standard "glueing" model. The algorithm is written in the dependently typed functional language of Martin-Löf type theory, and a correctness proof is given directly inside this language. Furthermore, we show how to modify the proof of the correctness of the NBE-algorithm to a proof of weak normalization and Church-Rosser for the usual notion of reduction for typed combinatory logic. Finally, we show how to implement this NBE-algorithm in Standard ML.

We begin Section 3 by discussing the suitability of the combinatory NBE-algorithm for the purpose of partial evaluation. We identify some of its shortcomings, such as the need for extensionality. Then we present the pure NBE algorithm for the typed lambda calculus with β and η conversion. This algorithm employs a non-standard interpretation of base types as term families.

In Section 4 we turn to type-directed partial evaluation, by which we mean the application of NBE to partial evaluation in more realistic programming languages. To this end we extend the pure typed lambda calculus with constants for arithmetic operations, conditionals, and fixed point computations yielding a version of the functional language PCF. In order to perform partial evaluation for this language we need to combine the pure NBE algorithm with the idea of off-line partial evaluation. We therefore introduce a notion of binding times, that allows us to separate occurrences of constant symbols in source programs into "static" and "dynamic" ones: the former are eliminated by the partial-evaluation process, while the latter remain in the residual code.

Finally, in Section 5, we show how the normalization technique is adapted from a semantic notion of equivalence based on $\beta\eta$-equality to one based on equality in the computational λ-calculus, which is the appropriate setting for call-by-value languages with computational effects.

Background reading. The reader of these notes is assumed to have some prior knowledge about lambda calculus and combinators, partial evaluation, functional programming, and semantics of programming languages, and we therefore list some links and reference works which are suitable for background reading.

For sections 2 and 3 we assume a reader familiar with combinatory logic and lambda calculus, including their relationship, type systems, models, and the notions of reduction and conversion. We recommend the following reference books as background reading. Note of course, that only a few of the basic concepts described in these books will be needed.

- Lecture notes on functional programming by Paulson [Pau00].
- Several reference articles by Barendregt on lambda calculi and functional programming [Bar77,Bar90,Bar92].

We use both dependent type theory and Standard ML (SML) as implementation languages, and assume that the reader is familiar with these languages. Knowledge of other typed functional languages such as OCAML or Haskell is of course also useful.

- Tutorials on Standard ML can be found at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/literature.html#tutorials`.
- For background reading about OCAML, see the chapter on *Objective CAML* by Didier Rémy in this volume.
- For background reading about Haskell, see references in the chapter on *Monads and Effects* by Nick Benton, John Hughes, and Eugenio Moggi in this volume.
- For background reading about dependent types, see the chapter on *Dependent Types in Programming* by Gilles Barthe and Thierry Coquand in this volume.

For Sections 4 and 5, we also assume some knowledge of domains and continuous functions, monads, and continuations.

- A good introduction to domains and their use in denotational semantics is Winskel's textbook [Win93].
- For background reading about monads, see the chapter on *Monads and Effects* by Nick Benton, John Hughes, and Eugenio Moggi in this volume.
- Three classic articles on continuations are by Reynolds [Rey72], Strachey and Wadsworth [SW74], and Plotkin [Plo75], although we will not make use of any specific results presented in those references.

Previous knowledge of some of the basic ideas from partial evaluation, including the notion of binding time is also useful. Some references:

- The standard textbook by Jones, Gomard, and Sestoft [JGS93].
- Tutorial notes on partial evaluation by Consel and Danvy [CD93a].
- Lecture notes on type-directed partial evaluation by Danvy [Dan98].

Acknowledgments. We gratefully acknowledge the contributions of our APPSEM 2000 co-lecturer Olivier Danvy to these notes, as well as the feedback and suggestions from the other lecturers and students at the meeting.

2 Normalization by evaluation for combinators

2.1 Combinatory System T

In this section we use a combinatory version of Gödel System T of primitive recursive functionals. In addition to the combinators K and S, this language has combinators ZERO for the number 0, SUCC for the successor function, and REC for primitive recursion.

The set of types is defined by

$$\vdash \mathsf{nat}\ \mathit{type} \qquad \frac{\vdash \tau_1\ \mathit{type} \quad \vdash \tau_2\ \mathit{type}}{\vdash \tau_1 \dot{\rightarrow} \tau_2\ \mathit{type}}$$

The typing rules for terms are

$$\frac{\vdash_{\mathrm{CL}} E : \tau_1 \dot{\to} \tau_2 \qquad \vdash_{\mathrm{CL}} E' : \tau_1}{\vdash_{\mathrm{CL}} E{\cdot}E' : \tau_2}$$

$$\begin{array}{rcl}
\vdash_{\mathrm{CL}} \mathsf{K}_{\tau_1\tau_2} & : & \tau_1 \dot{\to} \tau_2 \dot{\to} \tau_1 \\
\vdash_{\mathrm{CL}} \mathsf{S}_{\tau_1\tau_2\tau_3} & : & (\tau_1 \dot{\to} \tau_2 \dot{\to} \tau_3) \dot{\to} (\tau_1 \dot{\to} \tau_2) \dot{\to} \tau_1 \dot{\to} \tau_3 \\
\vdash_{\mathrm{CL}} \mathsf{ZERO} & : & \mathsf{nat} \\
\vdash_{\mathrm{CL}} \mathsf{SUCC} & : & \mathsf{nat} \dot{\to} \mathsf{nat} \\
\vdash_{\mathrm{CL}} \mathsf{REC}_\tau & : & \tau \dot{\to} (\mathsf{nat} \dot{\to} \tau \dot{\to} \tau) \dot{\to} \mathsf{nat} \dot{\to} \tau
\end{array}$$

We will often drop the subscripts of K, S, and REC, when they are clear from the context. Furthermore, we let $\mathrm{T}(\tau) = \{E \mid \vdash_{\mathrm{CL}} E : \tau\}$.

We now introduce the relation $\mathtt{conv}$ of convertibility of combinatory terms. We usually write $\vdash_{\mathrm{CL}} E = E'$ for $E \,\mathtt{conv}\, E'$. It is the least equivalence relation closed under the following rules

$$\frac{\vdash_{\mathrm{CL}} F = F' \qquad \vdash_{\mathrm{CL}} E = E'}{\vdash_{\mathrm{CL}} F{\cdot}E = F'{\cdot}E'}$$

$$\vdash_{\mathrm{CL}} \mathsf{K}_{\tau_1\tau_2}{\cdot}E{\cdot}E' = E$$

$$\vdash_{\mathrm{CL}} \mathsf{S}_{\tau_1\tau_2\tau_3}{\cdot}G{\cdot}F{\cdot}E = G{\cdot}E{\cdot}(F{\cdot}E)$$

$$\vdash_{\mathrm{CL}} \mathsf{REC}_\tau{\cdot}E{\cdot}F{\cdot}\mathsf{ZERO} = E$$

$$\vdash_{\mathrm{CL}} \mathsf{REC}_\tau{\cdot}E{\cdot}F{\cdot}(\mathsf{SUCC}{\cdot}N) = F{\cdot}N{\cdot}(\mathsf{REC}_\tau{\cdot}E{\cdot}F{\cdot}N)$$

2.2 Standard semantics

In the standard semantics we interpret a type τ as a set $[\![\tau]\!]$:

$$\begin{array}{rcl}
[\![\tau_1 \dot{\to} \tau_2]\!] & = & [\![\tau_1]\!] \to [\![\tau_2]\!] \\
[\![\mathsf{nat}]\!] & = & \mathbb{N}
\end{array}$$

where $\mathbb{N}$ is the set of natural numbers in the meta-language.

The interpretation $[\![E]\!] \in [\![\tau]\!]$ of an object $E \in \mathrm{T}(\tau)$ is defined by induction on E:

$$\begin{array}{rcl}
[\![\mathsf{K}_{\tau_1\tau_2}]\!] & = & \lambda x^{[\![\tau_1]\!]}.\, \lambda y^{[\![\tau_2]\!]}.\, x \\
[\![\mathsf{S}_{\tau_1\tau_2\tau_3}]\!] & = & \lambda g^{[\![\tau_1]\!]}.\, \lambda f^{[\![\tau_2]\!]}.\, \lambda x^{[\![\tau_3]\!]}.\, g\ x\ (f\ x) \\
[\![F{\cdot}E]\!] & = & [\![F]\!]\ [\![E]\!] \\
[\![\mathsf{ZERO}]\!] & = & 0 \\
[\![\mathsf{SUCC}]\!] & = & \mathrm{succ} \\
[\![\mathsf{REC}_\tau]\!] & = & \mathrm{rec}_{[\![\tau]\!]}
\end{array}$$

where succ $\in \mathbb{N} \to \mathbb{N}$ is the meta-language successor function

$$\mathrm{succ}\ n = n + 1$$

and $\mathrm{rec}_C \in C \to (\mathbb{N} \to C \to C) \to \mathbb{N} \to C$ is the meta-language primitive recursion operator defined by

$$\begin{array}{rcl} \mathrm{rec}_C\ e\ f\ 0 & = & e \\ \mathrm{rec}_C\ e\ f\ (\mathrm{succ}\ n) & = & f\ n\ (\mathrm{rec}_C\ e\ f\ n) \end{array}$$

Theorem 1. *If* $\vdash_{\mathrm{CL}} E = E'$ *then* $[\![E]\!] = [\![E']\!]$.

Proof. By induction on the proof that $\vdash_{\mathrm{CL}} E = E'$.

2.3 Normalization algorithm

The interpretation function into the standard model is not injective and hence cannot be inverted. For example, both $\mathsf{S}{\cdot}\mathsf{K}{\cdot}\mathsf{K}$ and $\mathsf{S}{\cdot}\mathsf{K}{\cdot}(\mathsf{S}{\cdot}\mathsf{K}{\cdot}\mathsf{K})$ denote identity functions. They are however not convertible, since they are distinct normal forms. (This follows from the normalization theorem below.)

We can construct a non-standard interpretation where the functions are interpreted as pairs of syntactic and semantic functions:

$$\begin{array}{rcl} [\![\mathsf{nat}]\!]^{\mathrm{Gl}} & = & \mathbb{N} \\ [\![\tau_1 \dot{\to} \tau_2]\!]^{\mathrm{Gl}} & = & \mathrm{T}(\tau_1 \dot{\to} \tau_2) \times ([\![\tau_1]\!]^{\mathrm{Gl}} \to [\![\tau_2]\!]^{\mathrm{Gl}}) \end{array}$$

We say that the model is constructed by "glueing" a syntactic and a semantic component, hence the notation $[\![-]\!]^{\mathrm{Gl}}$. (The glueing technique is also used in some approaches to partial evaluation [Asa02,Ruf93,SK01].)

Now we can write a function $\mathrm{reify}_\tau \in [\![\tau]\!]^{\mathrm{Gl}} \to \mathrm{T}(\tau)$ defined by

$$\begin{array}{rcl} \mathrm{reify}_{\mathsf{nat}}\ 0 & = & \mathsf{ZERO} \\ \mathrm{reify}_{\mathsf{nat}}\ (\mathrm{succ}\ n) & = & \mathsf{SUCC}{\cdot}(\mathrm{reify}_{\mathsf{nat}}\ n) \\ \mathrm{reify}_{\tau_1 \dot{\to} \tau_2}\ \langle F, f\rangle & = & F \end{array}$$

and which inverts the interpretation function $[\![-]\!]^{\mathrm{Gl}}_\tau \in \mathrm{T}(\tau) \to [\![\tau]\!]^{\mathrm{Gl}}$:

$$\begin{array}{rcl} [\![\mathsf{K}]\!]^{\mathrm{Gl}} & = & \langle \mathsf{K}, \lambda p.\, \langle \mathsf{K}{\cdot}(\mathrm{reify}\ p), \lambda q.\, p\rangle\rangle \\ [\![\mathsf{S}]\!]^{\mathrm{Gl}} & = & \langle \mathsf{S}, \lambda p.\, \langle \mathsf{S}{\cdot}(\mathrm{reify}\ p), \lambda q.\, \langle \mathsf{S}{\cdot}(\mathrm{reify}\ p){\cdot}(\mathrm{reify}\ q), \\ & & \qquad \lambda r.\, \mathrm{appsem}\ (\mathrm{appsem}\ p\ r)(\mathrm{appsem}\ q\ r)\rangle\rangle\rangle \\ [\![F{\cdot}E]\!]^{\mathrm{Gl}} & = & \mathrm{appsem}\ [\![F]\!]^{\mathrm{Gl}}\ [\![E]\!]^{\mathrm{Gl}} \\ [\![\mathsf{ZERO}]\!]^{\mathrm{Gl}} & = & 0 \\ [\![\mathsf{SUCC}]\!]^{\mathrm{Gl}} & = & \langle \mathsf{SUCC}, \mathrm{succ}\rangle \\ [\![\mathsf{REC}]\!]^{\mathrm{Gl}} & = & \langle \mathsf{REC}, \lambda p.\, \langle \mathsf{REC}{\cdot}(\mathrm{reify}\ p), \lambda q.\, \langle \mathsf{REC}{\cdot}(\mathrm{reify}\ p){\cdot}(\mathrm{reify}\ q), \\ & & \qquad \mathrm{rec}\ p\ (\lambda n r.\, \mathrm{appsem}\ (\mathrm{appsem}\ q\ n)\ r)\rangle\rangle\rangle \end{array}$$

We have here omitted type labels in lambda abstractions, and used the following application operator in the model:

$$\text{appsem } \langle F, f \rangle \ q \ = \ f \ q$$

It follows that the conversion rules are satisfied in this model, for example

$$[\![\mathsf{K} \cdot E \cdot E']\!]^{\mathrm{Gl}} = [\![E]\!]^{\mathrm{Gl}}$$

Theorem 2. *If* $\vdash_{\mathrm{CL}} E = E'$ *then* $[\![E]\!]^{\mathrm{Gl}} \ = \ [\![E']\!]^{\mathrm{Gl}}$.

The normal form function can now be defined by

$$\text{norm } E \ = \ \text{reify } [\![E]\!]^{\mathrm{Gl}}$$

Corollary 1. *If* $\vdash_{\mathrm{CL}} E = E'$ *then* norm $E \ = \ $ norm E'.

Theorem 3. $\vdash_{\mathrm{CL}} E = \text{norm } E$*, that is,* reify *is a left inverse of* $[\![-]\!]^{\mathrm{Gl}}$.

Proof. We use *initial algebra semantics* to structure our proof. A model of Gödel System T is a typed combinatory algebra extended with operations for interpreting ZERO, SUCC, and REC, such that the two equations for primitive recursion are satisfied. The syntactic algebra $\mathrm{T}(\tau)/\texttt{conv}$ of terms under convertibility is an initial model. The glueing model here is another model. The interpretation function $[\![-]\!]^{\mathrm{Gl}}_{\tau} \in \mathrm{T}(\tau)/\texttt{conv} \rightarrow [\![\tau]\!]^{\mathrm{Gl}}$ is the unique homomorphism from the initial model: if we could prove that $\text{reify}_{\tau} \in [\![\tau]\!]^{\mathrm{Gl}} \rightarrow \mathrm{T}(\tau)/\texttt{conv}$ also is a homomorphism, then it would follow that $\text{norm}_{\tau} \in \mathrm{T}(\tau)/\texttt{conv} \rightarrow \mathrm{T}(\tau)/\texttt{conv}$, the composition of $[\![\ ]\!]^{\mathrm{Gl}}_{\tau}$ and reify_{τ}, is also a homomorphism. Hence it must be equal to the identity homomorphism, and hence $\vdash_{\mathrm{CL}} E = \text{norm } E$.

But reify does not preserve application. However, we can construct a submodel of the non-standard model, such that the restriction of reify_{τ} to this submodel is a homomorphism. We call this the *glued* submodel; this construction is closely related to the glueing construction in category theory, see Lafont [Laf88, Appendix A].

In the submodel we require that a value $p \in [\![\tau]\!]^{\mathrm{Gl}}$ satisfies the property $\mathrm{Gl}_{\tau} \ p$ defined by induction on the type τ:

- $\mathrm{Gl}_{\mathsf{nat}} \ n$ holds for all $n \in \mathbb{N}$.
- $\mathrm{Gl}_{\tau_1 \rightarrow \tau_2} \ q$ holds *iff* for all $p \in [\![\tau_1]\!]^{\mathrm{Gl}}$ if $\mathrm{Gl}_{\tau_1} \ p$ then $\mathrm{Gl}_{\tau_2}(\text{appsem } q \ p)$ and $\vdash_{\mathrm{CL}} (\text{reify } q) \cdot (\text{reify } p) = \text{reify } (\text{appsem } q \ p)$

Notice that this construction can be made from any model, and not only the term model. In this way we can define the normalization function abstractly over any initial algebra for our combinatory system T.

Lemma 1. *The glued values* $\{ p \in [\![\tau_1]\!]^{\mathrm{Gl}} \mid \mathrm{Gl}_{\tau_1} p \}$ *form a model of Gödel System T.*

Proof: We show the case of K, and leave the other cases to the reader.

Case K. We wish to prove

$$\mathrm{Gl}_{\tau_1 \dot\to \tau_2 \dot\to \tau_1} \ \langle \mathsf{K}, \lambda p.\, \langle \mathsf{K}\cdot(\mathrm{reify}\ p), \lambda q.\, p\rangle\rangle$$

But this property follows immediately by unfolding the definition of $\mathrm{Gl}_{\tau_1 \dot\to \tau_2 \dot\to \tau_1}$ and using

$$\vdash_{\mathrm{CL}} \mathsf{K}\cdot(\mathrm{reify}\ p)\cdot(\mathrm{reify}\ q) = \mathrm{reify}\ p$$

Lemma 2. reify *is a homomorphism from the algebra of glued values to the term algebra.*

The definition of glued value is such that reify commutes with syntactic application. The other cases are immediate from the definition.

It now follows that norm is an identity homomorphism as explained above.

Corollary 2. $\vdash_{\mathrm{CL}} E = E'$ *iff* norm $E =$ norm E'.

Proof. If norm $E =$ norm E' then $\vdash_{\mathrm{CL}} E = E'$, by theorem 4. The reverse implication is theorem 3.

2.4 Weak normalization and Church-Rosser

We end this section by relating the "reduction-free" approach and the standard, reduction-based approach to normalization. We thus need to introduce the binary relation `red` of reduction (in zero or more steps) for our combinatory version of Gödel System T. This relation is inductively generated by exactly the same rules as convertibility, except that the rule of symmetry is omitted.

We now prove *weak* normalization by modifying the glueing model and replacing $\vdash_{\mathrm{CL}} - = -$ in the definition of Gl by `red`:

- $\mathrm{Gl}_{\mathsf{nat}}\ n$ holds for all $n \in \mathbb{N}$.
- $\mathrm{Gl}_{\tau_1 \dot\to \tau_2}\ q$ holds *iff* for all $p \in [\![\tau_1]\!]^{\mathrm{Gl}}$ if $\mathrm{Gl}_{\tau_1}\ p$ then $\mathrm{Gl}_{\tau_2}(\mathrm{appsem}\ q\ p)$ and $(\mathrm{reify}\ q)\cdot(\mathrm{reify}\ p)$ `red` $(\mathrm{reify}\ (\mathrm{appsem}\ q\ p))$

Theorem 4. *Weak normalization:* E `red` norm E *and* norm E *is irreducible.*

Proof. The proof of $\vdash_{\mathrm{CL}} E =$ norm E is easily modified to a proof that E `red` norm E.

It remains to prove that norm E is a normal form (an irreducible term). A normal natural number is built up by SUCC and ZERO. Normal function terms are combinators standing alone or applied to insufficiently many normal arguments. If we let $\vdash^{\mathrm{nf}}_{\mathrm{CL}} E : \tau$ mean that E is a normal form of type τ we can inductively define it by the following rules:

$$\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{ZERO} : \mathsf{nat}$$

$$\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{SUCC} : \mathsf{nat} \dot\to \mathsf{nat} \qquad \frac{\vdash^{\mathrm{nf}}_{\mathrm{CL}} E : \mathsf{nat}}{\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{SUCC}\cdot E : \mathsf{nat}}$$

$$\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{K} : \tau_1 \dot{\rightarrow} \tau_2 \dot{\rightarrow} \tau_1 \qquad \frac{\vdash^{\mathrm{nf}}_{\mathrm{CL}} E : \tau_1}{\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{K}{\cdot}E : \tau_2 \dot{\rightarrow} \tau_1}$$

$$\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{S} : (\tau_1 \dot{\rightarrow} \tau_2 \dot{\rightarrow} \tau_3) \dot{\rightarrow} (\tau_1 \dot{\rightarrow} \tau_2) \dot{\rightarrow} \tau_1 \dot{\rightarrow} \tau_3$$

$$\frac{\vdash^{\mathrm{nf}}_{\mathrm{CL}} G : \tau_1 \dot{\rightarrow} \tau_2 \dot{\rightarrow} \tau_3}{\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{S}{\cdot}G : (\tau_1 \dot{\rightarrow} \tau_2) \dot{\rightarrow} \tau_1 \dot{\rightarrow} \tau_3}$$

$$\frac{\vdash^{\mathrm{nf}}_{\mathrm{CL}} G : \tau_1 \dot{\rightarrow} \tau_2 \dot{\rightarrow} \tau_3 \qquad \vdash^{\mathrm{nf}}_{\mathrm{CL}} F : \tau_1 \dot{\rightarrow} \tau_2}{\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{S}{\cdot}G{\cdot}F : \tau_1 \dot{\rightarrow} \tau_3}$$

$$\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{REC} : \tau \dot{\rightarrow} (\mathsf{nat} \dot{\rightarrow} \tau \dot{\rightarrow} \tau) \dot{\rightarrow} \mathsf{nat} \dot{\rightarrow} \tau$$

$$\frac{\vdash^{\mathrm{nf}}_{\mathrm{CL}} E : \tau}{\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{REC}{\cdot}E : (\mathsf{nat} \dot{\rightarrow} \tau \dot{\rightarrow} \tau) \dot{\rightarrow} \mathsf{nat} \dot{\rightarrow} \tau}$$

$$\frac{\vdash^{\mathrm{nf}}_{\mathrm{CL}} E : \tau \qquad \vdash^{\mathrm{nf}}_{\mathrm{CL}} F : \mathsf{nat} \dot{\rightarrow} \tau \dot{\rightarrow} \tau}{\vdash^{\mathrm{nf}}_{\mathrm{CL}} \mathsf{REC}{\cdot}E{\cdot}F : \mathsf{nat} \dot{\rightarrow} \tau}$$

Let $\mathrm{T}^{\mathrm{nf}}(\tau) = \{E \mid \vdash^{\mathrm{nf}}_{\mathrm{CL}} E : \tau\}$ be the set of normal forms of type τ. If we redefine

$$[\![\tau_1 \dot{\rightarrow} \tau_2]\!]^{\mathrm{Gl}} \;=\; \mathrm{T}^{\mathrm{nf}}(\tau_1 \dot{\rightarrow} \tau_2) \times ([\![\tau_1]\!]^{\mathrm{Gl}} \rightarrow [\![\tau_2]\!]^{\mathrm{Gl}})$$

then we can verify that reify and norm have the following types

$$\begin{aligned} \mathrm{reify}_\tau &\in [\![\tau]\!]^{\mathrm{Gl}} \rightarrow \mathrm{T}^{\mathrm{nf}}(\tau) \\ \mathrm{norm}_\tau &\in \mathrm{T}(\tau) \rightarrow \mathrm{T}^{\mathrm{nf}}(\tau) \end{aligned}$$

and hence $\mathrm{norm}_\tau E \in \mathrm{T}^{\mathrm{nf}}(\tau)$ for $E \in \mathrm{T}(\tau)$.

Corollary 3. *Church-Rosser: if E `red` E' and E `red` E'' then there exists an E''' such that E' `red` E''' and E'' `red` E'''.*

Proof. It follows that $\vdash_{\mathrm{CL}} E' = E''$ and hence by theorem 3 that norm $E' =$ norm E''. Let $E''' =$ norm $E' =$ norm E'' and hence E' `red` E''' and E'' `red` E'''.

2.5 The normalization algorithm in Standard ML

We now show a sample implementation of the above algorithm in a conventional functional language.[3] We begin by defining the datatype `syn` of untyped terms:

```
datatype syn = S
             | K
             | APP of syn * syn
             | ZERO
             | SUCC
             | REC
```

[3] All the code in these notes is available electronically at `http://www.diku.dk/~andrzej/papers/NaPE-code.tar.gz`

We don't have dependent types in Standard ML so we have to collect all terms into this one type syn.

We implement the semantic natural numbers using SML's built-in type int of integers. The primitive recursion combinator can thus be defined as follows:

```
(* primrec : 'a * (int -> 'a -> 'a) -> int -> 'a *)

fun primrec (z, s)
    = let fun walk 0
              = z
            | walk n
              = let val p = n-1
                in s p (walk p)
                end
      in walk
      end
```

In order to build the non-standard interpretation needed for the normalization function, we introduce the following reflexive datatype sem of semantic values:

```
datatype sem = FUN of syn * (sem -> sem)
             | NAT of int
```

The function reify is implemented by

```
(* reify : sem -> syn *)

fun reify (FUN (syn, _))
    = syn
  | reify (NAT n)
    = let fun reify_nat 0
              = ZERO
            | reify_nat n
              = APP (SUCC, reify_nat (n-1))
      in reify_nat n
      end
```

Before writing the non-standard interpretation function we need some auxiliary semantic functions:

```
(* appsem : sem * sem -> sem
   succsem : sem -> sem
   recsem : 'a * (int -> 'a -> 'a) -> sem -> 'a *)

exception NOT_A_FUN

fun appsem (FUN (_, f), arg)
    = f arg
  | appsem (NAT _, arg)
    = raise NOT_A_FUN

exception NOT_A_NAT

fun succsem (FUN _)
    = raise NOT_A_NAT
  | succsem (NAT n)
    = NAT (n+1)

fun recsem (z, s) (FUN _)
    = raise NOT_A_NAT
  | recsem (z, s) (NAT n)
    = primrec (z, s) n
```

And thus we can write the non-standard interpretation function:

```
(* eval : syn -> sem *)

fun eval S
    = FUN (S,
           fn f => let val Sf = APP (S, reify f)
                   in FUN (Sf,
                           fn g => let val Sfg = APP (Sf, reify g)
                                   in FUN (Sfg,
                                           fn x
                                           => appsem (appsem (f, x),
                                                      appsem (g, x)))
                                   end)
                   end)
  | eval K
    = FUN (K,
           fn x => let val Kx = APP (K, reify x)
                   in FUN (Kx,
                           fn _ => x)
                   end)
  | eval (APP (e0, e1))
    = appsem (eval e0, eval e1)
  | eval ZERO
    = NAT 0
  | eval SUCC
    = FUN (SUCC,
           succsem)
  | eval REC
    = FUN (REC,
           fn z
           => let val RECz = APP (REC, reify z)
              in FUN (RECz,
                      fn s
                      => let val RECzs = APP (RECz, reify s)
                         in FUN (RECzs,
                                 recsem (z,
                                         fn n
                                         => fn c
                                            => appsem (appsem (s,
                                                               NAT n),
                                                       c)))
                         end)
              end)
```

Finally, the normalization function is

```
(* norm : syn -> syn *)
```

```
fun norm e
    = reify (eval e)
```

How do we know that the SML program is a correct implementation of the dependently typed (or "mathematical") normalization function? This is a non-trivial problem, since we are working in a language where we can write non-terminating well-typed programs. So, unlike before we cannot use the direct mathematical normalization proof, but have to resort to the operational or denotational semantics of SML. Note in particular the potential semantic complications of using the reflexive type `sem`.

Nevertheless, we claim that for any two terms `E`, `E' : syn` which represent elements of $\mathrm{T}(\tau)$, `norm E` and `norm E'` both terminate with identical values of `syn` iff `E` and `E'` represent convertible terms.

2.6 Exercises

Exercise 1. Extend NBE and its implementation to some or all of the following combinators:

```
      I x = x                              C f x y = f y x
  B f g x = f (g x)                          W f x = f x x
```

Exercise 2. Extend the datatype `sem` to have a double (that is, syntactic and semantic) representation of natural numbers, and modify NBE to cater for this double representation. Is either the simple representation or the double representation more efficient, and why?

Exercise 3. Because ML follows call by value, a syntactic witness is constructed for each intermediate value, even though only the witness of the final result is needed. How would you remedy that?

Exercise 4. Where else does ML's call by value penalize the implementation of NBE? How would you remedy that?

Exercise 5. Program a rewriting-based lambda calculus reducer and compare it to NBE in efficiency.

Exercise 6. Implement NBE in a language with dependent types such as Cayenne.

3 Normalization by evaluation for the $\lambda_{\beta\eta}$-calculus

In the next section, we shall see how normalization by evaluation can be exploited for the practical task of type-directed partial evaluation (TDPE) [Dan98]. TDPE is not based on the NBE-algorithm for combinators given in the previous section, but on the NBE-algorithm returning long $\beta\eta$-normal forms first presented by Berger and Schwichtenberg [BS91]. There are several reasons for this change:

Syntax. Most obviously, SK-combinators are far from a practical programming language. Although the algorithm extends directly to a more comprehensive set (e.g., SKBCI-combinators), we still want to express source programs in a more conventional lambda-syntax with variables. To use any combinator-based normalization algorithm, we would thus need to convert original programs to combinator form using bracket abstraction, and then either somehow evaluate combinator code directly, or convert it back to lambda-syntax (without undoing normality in the process – simply replacing the combinators with their definitions would not work!).

Thus, we prefer an algorithm that works on lambda-terms directly. As a consequence, however, we need to keep track of bound-variable names, and in particular avoid inadvertent clashes. While this is simple enough to do informally, we must be careful to express the process precisely enough to be analyzed, while keeping the implementation efficient.

Extensionality. As shown by Martin-Löf [ML75b] and by Coquand and Dybjer [CD93b], the glueing technique from the previous section can also be used for normalizing terms in a version of lambda-syntax. A problem with both variants, however, is the lack of extensionality. As we already mentioned, $\mathsf{S}{\cdot}\mathsf{K}{\cdot}\mathsf{K}$ and $\mathsf{S}{\cdot}\mathsf{K}{\cdot}(\mathsf{S}{\cdot}\mathsf{K}{\cdot}\mathsf{K})$ are both normal forms representing the identity function, which is somewhat unsatisfactory.

Even more problematically, these algorithms only compute *weak* normal forms. That is, the notion of conversion does not include (the combinatory analog of) the ξ-rule, which allows normalization under lambdas. Being able to reduce terms with free variables is a key requirement for partial evaluation. Consider, for example, the addition function in pseudo-lambda-syntax (with λ^* denoting bracket abstraction),

$$\begin{aligned} \mathit{add} &= \lambda^* m.\lambda^* n.\mathsf{REC}{\cdot}m{\cdot}(\lambda^* a.\lambda^* x.\mathsf{SUCC}{\cdot}x){\cdot}n \\ &= \lambda^* m.\mathsf{REC}{\cdot}m{\cdot}(\mathsf{K}{\cdot}\mathsf{SUCC}) \\ &= \mathsf{S}{\cdot}\mathsf{REC}{\cdot}(\mathsf{K}{\cdot}(\mathsf{K}{\cdot}\mathsf{SUCC})) \end{aligned}$$

$\mathit{add}{\cdot}m{\cdot}n$ applies the successor function n times to m. Given the reduction equations $\mathsf{REC}{\cdot}b{\cdot}f{\cdot}\mathsf{ZERO} = b$ and $\mathsf{REC}{\cdot}b{\cdot}f{\cdot}(\mathsf{SUCC}{\cdot}n) = f{\cdot}n{\cdot}(\mathsf{REC}{\cdot}b{\cdot}f{\cdot}n)$, we would hope that a partially applied function such as

$$\lambda^* m.\,\mathit{add}{\cdot}m{\cdot}(\mathsf{SUCC}{\cdot}(\mathsf{SUCC}{\cdot}\mathsf{ZERO})) = \mathsf{S}{\cdot}\mathit{add}{\cdot}(\mathsf{K}{\cdot}(\mathsf{SUCC}{\cdot}(\mathsf{SUCC}{\cdot}\mathsf{ZERO})))$$

could be normalized into $\lambda^* m.\mathsf{SUCC}\cdot(\mathsf{SUCC}\cdot m)$, i.e., eliminating the primitive recursion. But unfortunately, the combinatory term above is already in normal form with respect to the rewriting rules for S, K and REC, because all the combinators are unsaturated (i.e., applied to fewer arguments than their rewrite rules expect). Thus, we cannot unfold computations based on statically known data when the computation is also parameterized over unknown data.

It is fairly simple to extend the glueing-based normalization algorithms with top-level free variables; effectively, we just treat unknown inputs as additional, uninterpreted constants. With this extension the addition example goes through. However, the problem is not completely solved since we still do not simplify under *internal* lambdas in the program. For example, with

$$\mathit{mul} = \lambda^* m.\lambda^* n.\mathsf{REC}\cdot\mathsf{ZERO}\cdot(\lambda^* a.\lambda^* x.\mathit{add}\cdot x\cdot n)\cdot m\,,$$

we would want the open term $\mathit{mul}\cdot m\cdot(\mathsf{SUCC}\cdot(\mathsf{SUCC}\cdot\mathsf{ZERO}))$ to normalize to something like

$$\mathsf{REC}\cdot\mathsf{ZERO}\cdot(\lambda^* a.\lambda^* x.\mathsf{SUCC}\cdot(\mathsf{SUCC}\cdot x))\cdot m\,,$$

i.e., to eliminate at least the primitive recursion inside add; but again the necessary reductions will be blocked.

Native implementation. A final problem with the glueing-based algorithm is that the non-standard interpretation of terms differs significantly from the standard one for function spaces. This means that we have to construct a special-purpose evaluator; even if we already had an efficient standard evaluator for combinatory System T, we would not be able to use it directly for the task of normalization. The same problem appears for lambda-terms: we do have very efficient standard evaluators for functional programs, but we may not be able to use them if the interpretation of lambda-abstraction and application is seriously non-standard.

In this section, we present a variant of the Berger and Schwichtenberg NBE algorithm. (The main difference to the original is that we use a somewhat simpler scheme for avoiding variable clashes.) Again, we stress that the dimensions of syntax and convertibility are largely independent: one can also devise NBE-like algorithms for lambda-syntax terms based on β-conversion only [Mog92]. Likewise, it is also perfectly possible to consider $\beta\eta$-convertibility in a combinatory setting, especially for a different basis, such as categorical combinators [AHS95]. The following (obviously incomplete) table summarizes the situation:

	notion of conversion		
syntax	weak β	strong β	$\beta\eta$
combinators	[CD97]		[AHS95]
lambda-terms	[ML75b,CD93b]	[Mog92]	[BS91]

We should also mention that a main advantage with the glueing technique is that it extends smoothly to datatypes. We showed only how to treat natural numbers in the previous section, but the approach extends smoothly to arbitrary strictly positive datatypes such as the datatype of Brouwer ordinals [CD97]. For TDPE it is of course essential to deal with functions on datatypes, and in Section 4 we show how to deal with this problem by combining the idea of binding-time separation with normalization by evaluation.

3.1 The setting: simply typed lambda calculus

For the purpose of presenting the algorithm, let us ignore for the moment constant symbols and concentrate on pure lambda-terms only. Accordingly, consider a fixed collection of base types b; the *simple types* τ are then generated from those by the rules

$$\frac{}{\vdash b\ \mathit{type}} \qquad \frac{\vdash \tau_1\ \mathit{type} \qquad \vdash \tau_2\ \mathit{type}}{\vdash \tau_1 \rightarrow \tau_2\ \mathit{type}}$$

For a typing context Δ, assigning simple types to variables, the well-typed lambda-terms over Δ, $\Delta \vdash E : \tau$, are inductively generated by the usual rules:

$$\frac{\Delta(x) = \tau}{\Delta \vdash x : \tau} \qquad \frac{\Delta, x\colon \tau_1 \vdash E : \tau_2}{\Delta \vdash \lambda x^{\tau_1}.E : \tau_1 \rightarrow \tau_2} \qquad \frac{\Delta \vdash E_1 : \tau_1 \rightarrow \tau_2 \qquad \Delta \vdash E_2 : \tau_1}{\Delta \vdash E_1 \cdot E_2 : \tau_2}$$

We write $E =_\alpha E'$ if E' can be obtained from E by a consistent, capture-avoiding renaming of bound variables. We introduce the notion of $\beta\eta$-convertibility as a judgment $\vdash E =_{\beta\eta} E'$, generated by the following rules, together with reflexivity, transitivity, symmetry, and α-conversion:

$$\frac{\vdash E_1 =_{\beta\eta} E_1' \qquad \vdash E_2 =_{\beta\eta} E_2'}{\vdash E_1 \cdot E_2 =_{\beta\eta} E_1' \cdot E_2'} \qquad \frac{\vdash E =_{\beta\eta} E'}{\vdash \lambda x.E =_{\beta\eta} \lambda x.E'}$$

$$\frac{}{\vdash (\lambda x.E_1)\cdot E_2 =_{\beta\eta} E_1[E_2/x]} \qquad \frac{}{\vdash \lambda x.E \cdot x =_{\beta\eta} E}\ {\scriptstyle (x \notin FV(E))}$$

Here $E_1[E_2/x]$ denotes the capture-avoiding substitution of E_2 for free occurrences of x in E_1 (which in general may require an initial α-conversion of E_1; the details are standard).

Let us also recall the usual notion of $\beta\eta$-long normal form for lambda-terms. In the typed setting, it is usually expressed using two mutually recursive judgments enumerating terms in *normal* and *atomic* (also known as *neutral*) forms:

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

$$\frac{\Delta(x) = \tau}{\Delta \vdash^{\mathrm{at}} x : \tau} \qquad \frac{\Delta \vdash^{\mathrm{at}} E_1 : \tau_1 \rightarrow \tau_2 \qquad \Delta \vdash^{\mathrm{nf}} E_2 : \tau_1}{\Delta \vdash^{\mathrm{at}} E_1 \cdot E_2 : \tau_2}$$

One can show that any well-typed lambda-term $\Delta \vdash E : \tau$ is $\beta\eta$-convertible to exactly one (up to α-conversion) term $\tilde{E}$ such that $\Delta \vdash^{\mathrm{nf}} \tilde{E} : \tau$.

3.2 An informal normalization function

The usual way of computing long $\beta\eta$-normal forms is to repeatedly perform β-reduction steps until no β-redexes remain, and then η-expand the result until all variables are applied to as many arguments as their type suggests. We now present an alternative way of computing such normal forms.

Let $\mathbf{V}$ be a set of variable names and $\mathbf{E}$ be some set of elements suitable for representing lambda-terms. More precisely, we assume that there exist injective functions with disjoint ranges,

$$\mathrm{VAR} \in \mathbf{V} \to \mathbf{E} \qquad \mathrm{LAM} \in \mathbf{V} \times \mathbf{E} \to \mathbf{E} \qquad \mathrm{APP} \in \mathbf{E} \times \mathbf{E} \to \mathbf{E}$$

Perhaps the simplest choice is to take $\mathbf{E}$ as the set of (open, untyped) syntactic lambda-terms. But we could also take $\mathbf{V} = \mathbf{E} = \mathbb{N}$ with some form of Gödel-coding. More practically, we could take $\mathbf{V} = \mathbf{E}$ as the set of ASCII strings or the set of Lisp/Scheme S-expressions. In particular, we do not require that $\mathbf{E}$ does not contain elements other than the representation of lambda-terms.

Remark 1. It would be possible to let $\mathbf{E}$ be an object-type-indexed set family, as in the previous section, rather than a single set. We will not pursue such an approach for two reasons, though. First, for practical applications, it is important that correctness of the algorithm can be established in a straightforward way even when it is expressed in a non-dependently typed functional language. And second, there are additional complications in expressing the normalization algorithm for the call-by-value setting in the next section in a dependent-typed setting. The problems have to do with the interaction between computational effects such as continuations with a dependent type structure; at the time of writing, we do not have a viable dependently-typed algorithm for that case.

In any case, we can define a representation function $\ulcorner - \urcorner$ from well-formed lambda-terms with variables from $\mathbf{V}$ to elements of $\mathbf{E}$ by

$$\ulcorner x \urcorner = \mathrm{VAR}\, x \qquad \ulcorner \lambda x^{\tau}.E \urcorner = \mathrm{LAM}\,\langle x, \ulcorner E \urcorner\rangle \qquad \ulcorner E_1 \cdot E_2 \urcorner = \mathrm{APP}\,\langle \ulcorner E_1 \urcorner, \ulcorner E_2 \urcorner\rangle$$

(Note that we do not include the type tags for variables in representations of lambda-abstraction; the extension to do this is completely straightforward. See Exercise 7.) Because of the injectivity and disjointness requirements, for any $e \in \mathbf{E}$, there is then at most one E such that $\ulcorner E \urcorner = e$.

We now want to construct a *residualizing* interpretation, such that from the residualizing meaning of a term, we can extract its normal form, like before. Moreover, we want to interpret function types as ordinary function spaces. A natural first try at such an interpretation would thus be to assign to every type τ a set $[\![\tau]\!]^{\mathrm{r}}$ as follows:

$$\begin{aligned} [\![b]\!]^{\mathrm{r}} &= \mathbf{E} \\ [\![\tau_1 \to \tau_2]\!]^{\mathrm{r}} &= [\![\tau_1]\!]^{\mathrm{r}} \to [\![\tau_2]\!]^{\mathrm{r}} \end{aligned}$$

The interpretation of terms is then completely standard: let ρ be a Δ-environment, that is, a function assigning an element of $[\![\tau]\!]^{\mathrm{r}}$ to every x with

$\Delta(x) = \tau$. Then we define the residualizing meaning of a well-typed term $\Delta \vdash E : \tau$ as an element $[\![E]\!]^{\mathrm{r}}_{\rho} \in [\![\tau]\!]^{\mathrm{r}}$ by structural induction:

$$\begin{aligned}
[\![x]\!]^{\mathrm{r}}_{\rho} &= \rho\,(x) \\
[\![\underline{\lambda} x^{\tau}.\, E]\!]^{\mathrm{r}}_{\rho} &= \lambda a^{[\![\tau]\!]^{\mathrm{r}}}.\, [\![E]\!]^{\mathrm{r}}_{\rho[x \mapsto a]} \\
[\![E_1 \underline{\cdot} E_2]\!]^{\mathrm{r}}_{\rho} &= [\![E_1]\!]^{\mathrm{r}}_{\rho}\, [\![E_2]\!]^{\mathrm{r}}_{\rho}
\end{aligned}$$

This turns out to be a good attempt: for any type τ, it allows us to construct the following pair of functions, conventionally called *reification* and *reflection*:

$$\begin{aligned}
\downarrow_{\tau} &\in [\![\tau]\!]^{\mathrm{r}} \to \mathbf{E} \\
\downarrow_{b} &= \lambda t^{\mathbf{E}}.\, t \\
\downarrow_{\tau_1 \underline{\to} \tau_2} &= \lambda f^{[\![\tau_1]\!]^{\mathrm{r}} \to [\![\tau_2]\!]^{\mathrm{r}}}.\, \mathrm{LAM}\,\langle v, \downarrow_{\tau_2} (f\,(\uparrow_{\tau_1} (\mathrm{VAR}\, v)))\rangle \qquad (v \in \mathbf{V},\ \text{“fresh”}) \\
\uparrow_{\tau} &\in \mathbf{E} \to [\![\tau]\!]^{\mathrm{r}} \\
\uparrow_{b} &= \lambda e^{\mathbf{E}}.\, e \\
\uparrow_{\tau_1 \underline{\to} \tau_2} &= \lambda e^{\mathbf{E}}.\, \lambda a^{[\![\tau_1]\!]^{\mathrm{r}}}.\, \uparrow_{\tau_2} (\mathrm{APP}\,\langle e, \downarrow_{\tau_1} a\rangle)
\end{aligned}$$

Reification extracts a syntactic representation of a term from its residualizing semantics (as in the combinatory logic algorithm). Conversely, reflection wraps up a piece of syntax to make it act as an element of the corresponding type interpretation.

Together, these functions allow us to extract syntactic representations of closed lambda-terms from their denotations in the residualizing interpretation. For example,

$$\begin{aligned}
&\downarrow_{(b \underline{\to} b) \underline{\to} b \underline{\to} b} [\![\underline{\lambda} s.\, \underline{\lambda} z.\, s \underline{\cdot} (s \underline{\cdot} z)]\!]^{\mathrm{r}}_{\emptyset} = \downarrow_{(b \underline{\to} b) \underline{\to} b \underline{\to} b} (\lambda \phi.\, \lambda a.\, \phi\,(\phi\, a)) \\
&\quad = \mathrm{LAM}\,\langle x_1, \downarrow_{b \underline{\to} b} ((\lambda \phi.\, \lambda a.\, \phi\,(\phi\, a))\,(\uparrow_{b \underline{\to} b} (\mathrm{VAR}\, x_1)))\rangle \\
&\quad = \mathrm{LAM}\,\langle x_1, \downarrow_{b \underline{\to} b} ((\lambda \phi.\, \lambda a.\, \phi\,(\phi\, a))\,(\lambda a.\, \mathrm{APP}\,\langle \mathrm{VAR}\, x_1, a\rangle))\rangle \\
&\quad = \mathrm{LAM}\,\langle x_1, \downarrow_{b \underline{\to} b} (\lambda a.\, \mathrm{APP}\,\langle \mathrm{VAR}\, x_1, \mathrm{APP}\,\langle \mathrm{VAR}\, x_1, a\rangle\rangle)\rangle \\
&\quad = \mathrm{LAM}\,\langle x_1, \mathrm{LAM}\,\langle x_2, (\lambda a.\, \mathrm{APP}\,\langle \mathrm{VAR}\, x_1, \mathrm{APP}\,\langle \mathrm{VAR}\, x_1, a\rangle\rangle)\,(\mathrm{VAR}\, x_2)\rangle\rangle \\
&\quad = \mathrm{LAM}\,\langle x_1, \mathrm{LAM}\,\langle x_2, \mathrm{APP}\,\langle \mathrm{VAR}\, x_1, \mathrm{APP}\,\langle \mathrm{VAR}\, x_1, \mathrm{VAR}\, x_2\rangle\rangle\rangle\rangle
\end{aligned}$$

where we have arbitrarily chosen the fresh variable names $x_1, x_2 \in \mathbf{V}$ in the definition of reification at function types. Note that all the equalities in this derivation express definitional properties of functional abstraction and application in our set-theoretic metalanguage, as distinct from formal convertibility in the object language.

Given this extraction property for normal forms, it is now easy to see that $\downarrow_{\tau} [\![-]\!]^{\mathrm{r}}$ must be a normalization function, because $\beta\eta$-convertible terms have the same semantic denotation. Thus, for example, we would have obtained the same syntactic result if we had started instead with $\downarrow [\![\underline{\lambda} s.\, (\underline{\lambda} r.\, \underline{\lambda} z.\, r \underline{\cdot} (s \underline{\cdot} z)) \underline{\cdot} (\underline{\lambda} x.\, s \underline{\cdot} x)]\!]^{\mathrm{r}}_{\emptyset}$.

3.3 Formalizing unique name generation

On closer inspection, our definition of reification above is mathematically unsatisfactory. The problem is the “v fresh” condition: what exactly does it mean?

Unlike such conditions as "x does not occur free in E", it is not even locally checkable whether a variable is fresh; freshness is a global property, defined with respect to a term that may not even be fully constructed yet.

Needless to say, having such a vague notion at the core of an algorithm is a serious impediment to any formal analysis; we need a more precise way of talking about freshness. The concept can in fact be characterized rigorously in a framework such as Fraenkel-Mostowski sets, and even made accessible to the programmer as a language construct [GP99]. However, such an approach removes us a level from a direct implementation in a traditional, widely available functional language.

Instead, we explicitly generate non-clashing variable names. It turns out that we can do so fairly simply, if instead of working with individual term representations, we work with families of α-equivalent representations. The families will have the property that it is easy to control which variable names may occur bound in any particular member of the family. (This numbering scheme was also used by Berger [Ber93] and is significantly simpler than that in the original presentation of the algorithm [BS91].)

Definition 1. *Let* $\{g_0, g_1, g_2, \dots\} \subseteq \mathbf{V}$ *be a countably infinite set of variable names. We then define the set of* term families,

$$\hat{\mathbf{E}} = \mathbb{N} \to \mathbf{E}$$

together with the wrapper functions

$$\begin{aligned}
\widehat{\mathrm{VAR}} \in \mathbf{V} \to \hat{\mathbf{E}} &= \lambda v.\lambda i.\mathrm{VAR}\, v\\
\widehat{\mathrm{LAM}} \in (\mathbf{V} \to \hat{\mathbf{E}}) \to \hat{\mathbf{E}} &= \lambda f.\lambda i.\mathrm{LAM}\,\langle g_i, f\, g_i\,(i+1)\rangle\\
\widehat{\mathrm{APP}} \in \hat{\mathbf{E}} \times \hat{\mathbf{E}} \to \hat{\mathbf{E}} &= \lambda\langle e_1, e_2\rangle.\lambda i.\mathrm{APP}\,\langle e_1\, i, e_2\, i\rangle
\end{aligned}$$

We can construct term families using only these wrappers, then apply the result to a starting index i_0 and obtain a concrete representative not using any g_i's with $i < i_0$ as bound variables. For example,

$$\begin{aligned}
&(\widehat{\mathrm{LAM}}\,(\lambda v_1.\widehat{\mathrm{LAM}}\,(\lambda v_2.\widehat{\mathrm{APP}}\,(\widehat{\mathrm{VAR}} v_2, \widehat{\mathrm{APP}}\,(\widehat{\mathrm{VAR}} v_2, \widehat{\mathrm{VAR}}\, v_1))))\,7 = \cdots\\
&\quad = \mathrm{LAM}\,\langle g_7, \mathrm{LAM}\,\langle g_8, \mathrm{APP}\,\langle \mathrm{VAR}\, g_7, \mathrm{APP}\,\langle \mathrm{VAR}\, g_7, \mathrm{VAR}\, g_8\rangle\rangle\rangle\rangle
\end{aligned}$$

In general, each bound variable will be named g_i where i is the sum of the starting index and the number of lambdas enclosing the binding location. (This naming scheme is sometimes known as *de Bruijn levels* – not to be confused with *de Bruijn indices*, which assign numbers to individual uses of variables, not to their introductions.)

We now take for the residualizing interpretation,

$$\begin{aligned}
[\![b]\!]^{\mathrm{r}} &= \hat{\mathbf{E}}\\
[\![\tau_1 \to \tau_2]\!]^{\mathrm{r}} &= [\![\tau_1]\!]^{\mathrm{r}} \to [\![\tau_2]\!]^{\mathrm{r}}
\end{aligned}$$

and define corresponding reification and reflection functions:

$$\begin{aligned}
\downarrow_\tau &\in [\![\tau]\!]^{\mathrm{r}} \to \hat{\mathbf{E}} \\
\downarrow_b &= \lambda t^{\hat{\mathbf{E}}}.t \\
\downarrow_{\tau_1 \dot{\to} \tau_2} &= \lambda f^{[\![\tau_1]\!]^{\mathrm{r}} \to [\![\tau_2]\!]^{\mathrm{r}}}.\widehat{\mathrm{LAM}}\,(\lambda v^{\mathbf{V}}.\downarrow_{\tau_2}(f\,(\uparrow_{\tau_1}(\widehat{\mathrm{VAR}}\,v)))) \\
\uparrow_\tau &\in \hat{\mathbf{E}} \to [\![\tau]\!]^{\mathrm{r}} \\
\uparrow_b &= \lambda e^{\hat{\mathbf{E}}}.e \\
\uparrow_{\tau_1 \dot{\to} \tau_2} &= \lambda e^{\hat{\mathbf{E}}}.\lambda a^{[\![\tau_1]\!]^{\mathrm{r}}}.\uparrow_{\tau_2}(\widehat{\mathrm{APP}}\,\langle e, \downarrow_{\tau_1} a\rangle)
\end{aligned}$$

Finally we can state,

Definition 2. *We define the normalization function* norm *as follows: For any well-typed closed term* $\vdash E : \tau$, $\mathrm{norm}\,E$ *is the unique* $\tilde{E}$ *(if it exists) such that* $\ulcorner \tilde{E} \urcorner = \downarrow_\tau [\![E]\!]^{\mathrm{r}}_{\emptyset}\,0$.

Theorem 5 (Correctness). *We formulate correctness of* norm *as three criteria:*

1. norm *is total and type preserving: for any* $\vdash E : \tau$, $\mathrm{norm}\,E$ *denotes a well-defined* $\tilde{E}$, *and* $\vdash \tilde{E} : \tau$. *Moreover,* $\tilde{E}$ *is in normal form,* $\vdash^{\mathrm{nf}} \tilde{E} : \tau$.
2. $\vdash \mathrm{norm}\,E =_{\beta\eta} E$.
3. *If* $\vdash E =_{\beta\eta} E'$ *then* $\mathrm{norm}\,E = \mathrm{norm}\,E'$.

Several approaches are possible for the proof. Perhaps the simplest proceeds as follows: If $\tilde{E}$ is already in normal form, then it is fairly simple to show that $\mathrm{norm}\,\tilde{E} =_\alpha \tilde{E}$. (This follows by structural induction on the syntax of normal forms; the only complication is keeping track of variable renamings.) Moreover, as we observed in the informal example with "magic" fresh variables, if $E =_{\beta\eta} \tilde{E}$ then $[\![E]\!]^{\mathrm{r}}_{\emptyset} = [\![\tilde{E}]\!]^{\mathrm{r}}_{\emptyset}$, and hence $\mathrm{norm}\,E = \mathrm{norm}\,\tilde{E} =_\alpha \tilde{E}$ directly by the definition of norm. All the claims of the theorem then follow easily from the (non-trivial) fact that every term is $\beta\eta$-equivalent to one in normal form.

3.4 Implementation

An SML implementation of the $\lambda_{\beta\eta}$-normalizer is shown in Figure 1. Note that, unlike in the combinatory case, the central function `eval` is essentially the same as in a standard interpreter for a simple functional language. In the next section, we will see how we can take advantage of this similarity, by replacing the custom-coded interpretation function with the native evaluator of a functional language.

```
datatype term =
  VAR of string | LAM of string * term | APP of term * term

type termh = int -> term

fun VARh v = fn i => VAR v
fun LAMh f =
      fn i => let val v = "x" ^ Int.toString i in LAM (v, f v (i+1)) end
fun APPh (e1, e2) = fn i => APP (e1 i, e2 i)

datatype sem =
  TM of termh | FUN of sem -> sem

fun eval (VAR x) r = r x
  | eval (LAM (x, t)) r =
      FUN (fn a => eval t (fn x' => if x' = x then a else r x'))
  | eval (APP (e1, e2)) r =
      let val FUN f = eval e1 r in f (eval e2 r) end

datatype tp =
  BASE of string | ARROW of tp * tp

fun reify (BASE _) (TM e) = e
  | reify (ARROW (t1, t2)) (FUN f) =
      LAMh (fn v => reify t2 (f (reflect t1 (VARh v))))
and reflect (BASE _) e = TM e
  | reflect (ARROW (t1, t2)) e =
      FUN (fn a => reflect t2 (APPh (e, reify t1 a)))

fun norm t e =
      reify t (eval e (fn x => raise Fail ("Unbound: " ^ x))) 0

val test =
    norm (ARROW (ARROW (BASE "a", BASE "b"), ARROW (BASE "a", BASE "b")))
         (LAM ("f", LAM ("x", APP (LAM ("y", APP (VAR "f", VAR "y")),
                                   APP (VAR "f", VAR "x")))));
(* val test =
     LAM ("x0",LAM ("x1",APP (VAR "x0",APP (VAR "x0",VAR "x1")))) : term *)
```

Fig. 1. An implementation of the $\lambda_{\beta\eta}$-normalizer in SML

4 Type-directed partial evaluation for call-by-name

In this section, we will see how the general idea of normalization by evaluation can be exploited for the practical task of type-directed partial evaluation (TDPE) of functional programs [Dan98]. The main issues addressed here are:

Interpreted constants. A problem with the NBE algorithm for the pure $\lambda_{\beta\eta}$-calculus given in the previous section is that it is not clear how to extend it to, for example, System T, where we need to deal with primitive constants such as primitive recursion. Clearly, we cannot expect to interpret nat standardly, as we did for the combinatory version of System T: we cannot expect that all extensionally indistinguishable functions from natural numbers to natural numbers have the same normal form.

To recover a simple notion of equivalence, we need to introduce an explicit notion of *binding times* in the programs. That is, we must distinguish clearly between the *static* subcomputations, which should be carried out during normalization, and the *dynamic* ones, that will only happen when the normalized program itself is executed.

An *offline binding-time annotation* allows us to determine which parts of the program are static, even without knowing the actual static values. We can then distinguish between static and dynamic *occurrences* of operators, with their associated different conversion rules. This distinction will allow us to re-introduce interpreted constants, including recursion.

Recursion. When we want to use NBE for conventional functional programs, we get an additional complication in the form of unrestricted recursion. While many uses of general recursion can be replaced with suitable notions of primitive recursion, this is unfortunately not the case for one of the primary applications of partial evaluation, namely programming-language interpreters: since the interpreted program may diverge, the interpreter cannot itself be a total function either.

In fact, normalization of programs with interpreted constants is often expressed more naturally with respect to a *semantic* notion of equivalence, rather than syntactic $\beta\eta$-convertibility with additional rules. And in particular, to properly account for general recursion, it becomes natural to consider domain-theoretic models for both the standard and the non-standard interpretations, rather than purely set-theoretic ones.

4.1 The setting: A domain-theoretic semantics of PCF

Our prototypical functional language has the following set of well-formed types, $\vdash_\Sigma \sigma\ \mathit{type}$:

$$\frac{b \in \Sigma}{\vdash_\Sigma b\ \mathit{type}} \qquad \frac{\vdash_\Sigma \sigma_1\ \mathit{type} \quad \vdash_\Sigma \sigma_2\ \mathit{type}}{\vdash_\Sigma \sigma_1 \dashrightarrow \sigma_2\ \mathit{type}}$$

where the signature Σ contains a collection of base types b. (Note that the change from τ to σ as a metavariable for object types, and Δ to Γ for typing contexts below, is deliberate, in preparation for the next section.) Although we limit ourselves to base and function types for conciseness, adding finite-product types would be straightforward.

Each base type b is equipped with a countable collection of *literals* (numerals, truth values, etc.) $\Xi(b)$. These represent the observable results of program execution.

A typing context Γ is a finite mapping of variable names to well-formed types over Σ. Then the well-typed Σ-terms over Γ, $\Gamma \vdash_\Sigma E : \sigma$, are much as before:

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

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

Here x ranges over a countable set of variables, and c over a set of function constants in Σ. Note that some constants, such as conditionals, are actually polymorphic families of constants. We must explicitly pick out the relevant instance using type subscripts. We say that a well-typed term is a *complete program* if it is closed and of base type.

Since we want to model general recursion, we use a domain-theoretic model instead of a simpler set-theoretic one. (However, it is possible to understand most of the following constructions by ignoring the order structure and continuity, and simply thinking of domains as sets; only the formal treatment of fixed pointss suffers from such a simplification.)

Accordingly, we say that 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, that is, a possibly bottomless cpo, usually discretely ordered. Then we can interpret every type phrase σ over Σ as a domain (pointed cpo):

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

(For any cpo A, we write $A_\bot = A \cup \{\bot\}$ for its lifting, where the additional element $\bot$ signifies divergence. The interpretation of function types is the usual continuous-function space. Since we only deal with continuous functions in the semantics, we generally omit the subscript c from now on.)

Then, 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 $\mathcal{C}(c_{\sigma_1,\ldots,\sigma_n}) \in [\![\Sigma(c_{\sigma_1,\ldots,\sigma_n})]\!]^{\mathcal{I}}$. For simplicity, we assume that $\mathcal{L}_b$ is surjective, that is, that every element of $\mathcal{B}(b)$ is denoted by a literal.

We interpret a typing assignment Γ as a labelled product:

$$[\![\Gamma]\!]^{\mathcal{I}} = \prod_{x \in \operatorname{dom} \Gamma} [\![\Gamma(x)]\!]^{\mathcal{I}}$$

(Note that this is now a finite product of domains, ordered pointwise, i.e., $\rho \sqsubseteq \rho'$ iff $\forall x \in \mathrm{dom}\, \Gamma.\, \rho\, x \sqsubseteq_{\Gamma(x)} \rho'\, x$.)

To make a smooth transition to the later parts, let us express the semantics of terms with explicit reference to the *lifting monad* $(-_\perp, \eta^\perp, \star^\perp)$, where $\eta^\perp\, a$ is the *inclusion* of $a \in A$ into $A_\perp$, and $t \star^\perp f$ is the *strict extension* of $f \in A \to B_\perp$ applied to $t \in A_\perp$, given by $\perp_A \star^\perp f = \perp_B$ and $(\eta^\perp\, a) \star^\perp f = f\, a$. We then give the semantics of a term $\Gamma \vdash E : \tau$ as a (total) continuous function $[\![E]\!]^{\mathcal{I}} \in [\![\Gamma]\!]^{\mathcal{I}} \to [\![\tau]\!]^{\mathcal{I}}$:

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

Finally, given an interpretation $\mathcal{I}$ of Σ we define the partial function returning the observable result of evaluating a complete program:

$$\mathrm{eval}^{\mathcal{I}} \in \{E \mid\ \vdash_\Sigma E : b\} \rightharpoonup \Xi(b)$$

by

$$\mathrm{eval}^{\mathcal{I}}\, E = \begin{cases} l & \text{if } [\![E]\!]^{\mathcal{I}}\, \emptyset = \eta^\perp\, (\mathcal{L}_b(l)) \\ \text{undefined} & \text{if } [\![E]\!]^{\mathcal{I}}\, \emptyset = \perp \end{cases}$$

where $\emptyset$ is the empty environment.

Definition 3 (standard static language). *We define a simple functional language (essentially PCF [Plo77]) by taking the signature Σ_{s} as follows. The base types are* int *and* bool*; the literals,* $\Xi(\mathsf{int}) = \{\ldots, \text{-}1, 0, 1, 2, \ldots\}$ *and* $\Xi(\mathsf{bool}) = \{\mathsf{true}, \mathsf{false}\}$*; and the constants,*

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

(We use infix notation for applications of binary operations, for example, $x + y$ instead of $+ \cdot x \cdot y$.)

Similarly, the standard interpretation of this signature is also as expected:

$$\begin{aligned}
\mathcal{B}_{\mathrm{s}}(\mathsf{bool}) &= \mathbb{B} = \{\mathrm{true}, \mathrm{false}\} \\
\mathcal{B}_{\mathrm{s}}(\mathsf{int}) &= \mathbb{Z} = \{\ldots, -1, 0, 1, 2, \ldots\} \\
\mathcal{C}_{\mathrm{s}}(\diamond) &= \lambda x^{\mathbb{Z}_\perp}.\, \lambda y^{\mathbb{Z}_\perp}.\, x \star^\perp \lambda n.\, y \star^\perp \lambda m.\, \eta^\perp\, (m \diamond n) \quad {\scriptstyle \diamond \in \{+,-,\times,=,<\}} \\
\mathcal{C}_{\mathrm{s}}(\mathsf{if}_\sigma) &= \lambda x^{\mathbb{B}_\perp}.\, \lambda a_1^{[\![\sigma]\!]}.\, \lambda a_2^{[\![\sigma]\!]}.\, x \star^\perp \lambda b.\, \mathrm{if}\ b\ \mathrm{then}\ a_1\ \mathrm{else}\ a_2 \\
\mathcal{C}_{\mathrm{s}}(\mathsf{fix}_\sigma) &= \lambda f^{[\![\sigma]\!] \to [\![\sigma]\!]}.\, \bigsqcup\nolimits_{i \in \omega} f^i \perp_{[\![\sigma]\!]}
\end{aligned}$$

(where the conditional if b then x else y *chooses between x and y based on the truth value b.)*

It is well known (computational adequacy of the denotational semantics for call-by-name evaluation [Plo77]) that with this interpretation, $\text{eval}^{\mathcal{I}_s}$ is computable.

Remark 2. This PCF semantics differs from the standard semantics of Haskell in that there is no extra lifting of function types; Haskell would have $[\![\tau_1 \dot{\to} \tau_2]\!] = ([\![\tau_1]\!] \to [\![\tau_2]\!])_\perp$. One can show that this makes no difference, as long as we cannot observe termination at higher types. That is, with the typing restriction on complete programs, our denotational semantics is actually computationally adequate whether we evaluate programs in PCF or in Haskell.

Note that being able to observe termination at higher types breaks the validity of η-conversion as a semantic equivalence ($\dot{\lambda} x. f \cdot x$ and f become operationally distinguishable when f can be replaced with a non-terminating term), for no clear gain in convenience or expressive power.

4.2 Binding-time separation and static normal forms

For partial evaluation, we distinguish between operations that can be performed knowing only the static input, and those that also require dynamic data. A particularly useful way of making this distinction is through an *off-line binding-time annotation (BTA)*, where knowledge about which arguments to a program's top-level function will be static, and which will be dynamic, is propagated throughout the program in a separate phase. Note that this can be done without knowing the actual *values* of the static arguments.

The annotation can either be performed by a program (so called binding-time analysis), or explicitly by the programmer as the program is written. The latter is quite practical when the usage pattern is fixed – for example, an interpreter may be specialized with respect to a program, but practically never with respect to the program's input data.

Traditional binding-time annotations for typed languages are often expressed in terms of two-level types [NN88], where types and type constructors (such as function spaces and products) are annotated as static or dynamic. Type-directed partial evaluation is unusual in that the binding-time annotations are expressed in an essentially standard type system, which allows the annotations to be verified by an ML type checker. It also means that the annotated programs remain directly executable.

Specifically, BTA in TDPE is performed by expressing the program as a term over a *binding-time separated signature*, where the declarations of base types and constants are divided into a static and a dynamic part. That is, $\Sigma = \Sigma_\text{s} \cup \Sigma_\text{d}$ where each of Σ_s and Σ_d is itself a signature. Following the tradition, 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, that is, $\Xi(\underline{b}) = \emptyset$. (If needed, they can be added as dynamic constants.) However, some base types will be *persistent*, that is, have both static and dynamic versions with the same intended meaning. In that case, we also include *lifting functions*

$$\$_b : \overline{b} \dot{\to} \underline{b}$$

in the dynamic signature.

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

$$\frac{\underline{b} \in \Sigma_{\mathrm{d}}}{\underline{b}\ \mathit{dtype}} \qquad \frac{\tau_1\ \mathit{dtype} \quad \tau_2\ \mathit{dtype}}{\tau_1 \dot{\to} \tau_2\ \mathit{dtype}}$$

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, ensuring that the type of every instance is fully dynamic, for example, $\Sigma_{\mathrm{d}}(\underline{\mathsf{if}}_\tau) = \underline{\mathsf{bool}} \dot{\to} \tau \dot{\to} \tau \dot{\to} \tau$. On the other hand, constants from Σ_{s} can be instantiated at both static and dynamic types.

We will always take the language from Definition 3 with the standard semantics $\mathcal{I}_{\mathrm{s}} = (\mathcal{B}_{\mathrm{s}}, \mathcal{L}, \mathcal{C}_{\mathrm{s}})$ 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 of the dynamic signature.

Example 1. Given the functional term

$$\mathit{power} : \iota \dot{\to} \iota \dot{\to} \iota \;\equiv\; \dot{\lambda} x^{\iota}.\, \mathsf{fix}_{\iota \dot{\to} \iota} \cdot (\dot{\lambda} p^{\iota \dot{\to} \iota}.\, \dot{\lambda} n^{\iota}.\, \mathsf{if}_{\iota} \cdot (n = 0) \cdot 1 \cdot (x \times p \cdot (n - 1)))$$

(abbreviating int as ι), we can binding-time annotate it in four different ways, depending on which arguments will be statically known:

$$\begin{aligned}
\mathit{power}_{ss} : \overline{\iota} \dot{\to} \overline{\iota} \dot{\to} \overline{\iota} \;&\equiv\; \dot{\lambda} x^{\overline{\iota}}.\, \overline{\mathsf{fix}}_{\overline{\iota} \dot{\to} \overline{\iota}} \cdot (\dot{\lambda} p^{\overline{\iota} \dot{\to} \overline{\iota}}.\, \dot{\lambda} n^{\overline{\iota}}.\, \overline{\mathsf{if}}_{\overline{\iota}} \cdot (n \,\overline{=}\, 0) \cdot 1 \cdot (x \,\overline{\times}\, p \cdot (n \,\overline{-}\, 1))) \\
\mathit{power}_{sd} : \overline{\iota} \dot{\to} \underline{\iota} \dot{\to} \underline{\iota} \;&\equiv\; \dot{\lambda} x^{\overline{\iota}}.\, \underline{\mathsf{fix}}_{\underline{\iota} \dot{\to} \underline{\iota}} \cdot (\dot{\lambda} p^{\underline{\iota} \dot{\to} \underline{\iota}}.\, \dot{\lambda} n^{\underline{\iota}}.\, \underline{\mathsf{if}}_{\underline{\iota}} \cdot (n \,\underline{=}\, \$ \cdot 0) \cdot (\$ \cdot 1) \cdot (\$ \cdot x \,\underline{\times}\, p \cdot (n \,\underline{-}\, \$ \cdot 1))) \\
\mathit{power}_{ds} : \underline{\iota} \dot{\to} \overline{\iota} \dot{\to} \underline{\iota} \;&\equiv\; \dot{\lambda} x^{\underline{\iota}}.\, \overline{\mathsf{fix}}_{\overline{\iota} \dot{\to} \underline{\iota}} \cdot (\dot{\lambda} p^{\overline{\iota} \dot{\to} \underline{\iota}}.\, \dot{\lambda} n^{\overline{\iota}}.\, \overline{\mathsf{if}}_{\underline{\iota}} \cdot (n \,\overline{=}\, 0) \cdot (\$ \cdot 1) \cdot (x \,\underline{\times}\, p \cdot (n \,\overline{-}\, 1))) \\
\mathit{power}_{dd} : \underline{\iota} \dot{\to} \underline{\iota} \dot{\to} \underline{\iota} \;&\equiv\; \lambda x^{\underline{\iota}}.\, \underline{\mathsf{fix}}_{\underline{\iota} \dot{\to} \underline{\iota}} \cdot (\dot{\lambda} p^{\underline{\iota} \dot{\to} \underline{\iota}}.\, \dot{\lambda} n^{\underline{\iota}}.\, \underline{\mathsf{if}}_{\underline{\iota}} \cdot (n \,\underline{=}\, \$ \cdot 0) \cdot (\$ \cdot 1) (x \,\underline{\times}\, p \cdot (n \,\underline{-}\, \$ \cdot 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.

Some of the usual concerns of binding-time annotation arise for TDPE as well: for example, an unannotated term such as $(d+3)+s$, where d is a dynamic variable and s a static one, must be annotated as $(x \underline{+} \$ \cdot 3) \underline{+} \$ \cdot s$. That is, neither of the additions can be performed even with knowledge of s's value. Had we instead written the term as $d + (3 + s)$, we could annotate it as $d \,\underline{+}\, \$ \cdot (3 \,\overline{+}\, s)$, allowing the second addition to be eliminated at specialization time. Such rewritings are called *binding-time improvements.*

In keeping with the idea that computations involving dynamic constants should be left in the normalized program, we introduce a new notion of program equivalence, compatible with any future interpretation of the dynamic operators:

Definition 4 (Static Equivalence). *We say that E and E' are statically equivalent (wrt. a given static interpretation $\mathcal{I}_s$ of Σ_s), written $\mathcal{I}_s \models E = E'$, if for every interpretation $\mathcal{I}_d$ of Σ_d, $[\![E]\!]^{\mathcal{I}_s \cup \mathcal{I}_d} = [\![E']\!]^{\mathcal{I}_s \cup \mathcal{I}_d}$.*

(More generally, one can imagine an intermediate between static and dynamic constants: we may consider equivalence in all interpretations of Σ_d satisfying some additional constraints [Fil01].)

Note that static equivalence includes not only full $\beta\eta$-convertibility, but also equalities involving static constants, such as $\mathcal{I}_s \models \overline{\mathsf{fix}}{\cdot}f = f{\cdot}(\overline{\mathsf{fix}}{\cdot}f)$. However, in the particular case where the static signature contains no term constants, one can show that static equivalence coincides with $\beta\eta$-convertibility. (This is known as Friedman's completeness theorem for the full continuous type frame [Mit96, Theorem 8.4.6].)

4.3 A residualizing interpretation

To normalize a term with respect to static equivalence, we first adjust our syntactic characterization of normal forms: we allow constants from the dynamic signature only, and all literals must appear as arguments to \$. That is, we add the following two rules for atomic forms:

$$\frac{\Sigma_d(c_{\tau_1,\ldots,\tau_n}) = \tau}{\Delta \vdash^{\mathrm{at}} c_{\tau_1,\ldots,\tau_n} : \tau} \qquad \frac{l \in \Xi(\overline{b})}{\Delta \vdash^{\mathrm{at}} \$_b{\cdot}l : \underline{b}}$$

Also, corresponding to our extension of the language, we assume that in addition to VAR, LAM, and APP, we have two further syntax-constructor functions (again with ranges disjoint from those of the others),

$$\mathrm{CST} \in \mathbf{V} \to \mathbf{E} \qquad \mathrm{LIT}_b \in \mathcal{B}_s(b) \to \mathbf{E} \text{ (for each } b \text{ in } \Sigma_d)$$

As we only need to represent programs in normal form, we extend the representation equations as follows:

$$\ulcorner c_{\tau_1,\ldots,\tau_n} \urcorner = \mathrm{CST}\, c \qquad \ulcorner \$_b{\cdot}l \urcorner = \mathrm{LIT}_b\,(\mathcal{L}_b(l))$$

(For simplicity, we assume that elements of $\mathbf{V}$ can also be used to represent constant names. Also, we omit type tags in the generated representation of constants; like for lambda-abstraction, adding them is straightforward.)

Moreover, since we are working with domains, we need to take into account the possibility of nontermination at normalization time; that is, the reification function may need to return a $\perp$-result. Thus we take

$$\hat{\mathbf{E}} = \mathbb{Z}_\perp \to \mathbf{E}_\perp$$

where $\mathbf{E}$ is our set of term representations viewed as a discrete cpo. Elements of $\hat{\mathbf{E}}$ are ordered pointwise, that is $e \sqsubseteq e'$ iff for all $i \in \mathbb{Z}_\perp$, $e\,i = \perp$ or $e\,i = e'\,i$. For the purpose of the semantic presentation, we could have used $\mathbb{N}$ instead of

$\mathbb{Z}_\perp$ ($\hat{\mathbf{E}}$ would still be a pointed cpo with that choice), but we will make use of the revised definition in Section 4.4: unlike $\mathbb{N} \to \mathbf{E}_\perp$, $\mathbb{Z}_\perp \to \mathbf{E}_\perp$ is actually the denotation of a type in our standard static signature.

Correspondingly, we define our new wrapper functions taking lifting into account:

$$\begin{aligned}
\widehat{\mathrm{LIT}}_b \in \mathcal{B}_\mathrm{s}(b)_\perp \to \hat{\mathbf{E}} &= \lambda d.\, \lambda i.\, d \star^\perp \lambda n.\, \eta^\perp\, (\mathrm{LIT}_b\, n) \\
\widehat{\mathrm{CST}} \in \mathbf{V}_\perp \to \hat{\mathbf{E}} &= \lambda d.\, \lambda i.\, d \star^\perp \lambda v.\, \eta^\perp\, (\mathrm{CST}\, v) \\
\widehat{\mathrm{VAR}} \in \mathbf{V}_\perp \to \hat{\mathbf{E}} &= \lambda d.\, \lambda i.\, d \star^\perp \lambda v.\, \eta^\perp\, (\mathrm{VAR}\, v) \\
\widehat{\mathrm{LAM}} \in (\mathbf{V}_\perp \to \hat{\mathbf{E}}) \to \hat{\mathbf{E}} &= \\
&\lambda f.\, \lambda i.\, i \star^\perp \lambda j.\, f\, (\eta^\perp\, g_j)\, (\eta^\perp\, (j+1)) \star^\perp \lambda l.\, \eta^\perp\, (\mathrm{LAM}\, \langle g_j, l\rangle) \\
\widehat{\mathrm{APP}} \in \hat{\mathbf{E}} \times \hat{\mathbf{E}} \to \hat{\mathbf{E}} &= \lambda \langle e_1, e_2\rangle.\, \lambda i.\, e_1\, i \star^\perp \lambda l_1.\, e_2\, i \star^\perp \lambda l_2.\, \eta^\perp\, (\mathrm{APP}\, \langle l_1, l_2\rangle)
\end{aligned}$$

For the residualizing interpretation on types, we now specify

$$\mathcal{B}^\mathrm{r}_\mathrm{d}(\underline{b}) = \hat{\mathbf{E}}$$

Together with the standard interpretation of the static base types, this determines the semantics of any type σ over $\Sigma_\mathrm{s} \cup \Sigma_\mathrm{d}$.

The reification functions can be written exactly as before: for any dynamic type τ, we define a pair of continuous functions by induction on the structure of τ:

$$\begin{aligned}
\downarrow_\tau &\in [\![\tau]\!]^{\mathcal{I}_\mathrm{s} \cup \mathcal{I}^\mathrm{r}_\mathrm{d}} \to \hat{\mathbf{E}} \\
\downarrow_{\underline{b}} &= \lambda t.\, t \\
\downarrow_{\tau_1 \underline{\to} \tau_2} &= \lambda f.\, \widehat{\mathrm{LAM}}\, (\lambda v.\, \downarrow_{\tau_2} (f\, (\uparrow_{\tau_1} (\widehat{\mathrm{VAR}}\, v)))) \\
\uparrow_\tau &\in \hat{\mathbf{E}} \to [\![\tau]\!]^{\mathcal{I}_\mathrm{s} \cup \mathcal{I}^\mathrm{r}_\mathrm{d}} \\
\uparrow_{\underline{b}} &= \lambda e.\, e \\
\uparrow_{\tau_1 \underline{\to} \tau_2} &= \lambda e.\, \lambda a.\, \uparrow_{\tau_2} (\widehat{\mathrm{APP}}\, (e, \downarrow_{\tau_1} a))
\end{aligned}$$

Finally, we can construct the residualizing interpretation of terms. Again, we give only the interpretation of Σ_d's term constants; the semantics of lambda-abstraction and application are fixed by the semantic framework. We take:

$$\mathcal{C}^\mathrm{r}_\mathrm{d}(\underline{c}_{\tau_1,\ldots,\tau_n}) = \uparrow_{\Sigma_\mathrm{d}(\underline{c}_{\tau_1,\ldots,\tau_n})} (\widehat{\mathrm{CST}}\, c) \qquad \mathcal{C}^\mathrm{r}_\mathrm{d}(\$_b) = \widehat{\mathrm{LIT}}_b$$

That is, a general dynamic constant is simply interpreted as the reflection of its 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.)

Finally, we are again ready to define the normalization function:

Definition 5. *For any dynamic type τ, define the auxiliary function*

$$\mathrm{reify}_\tau \in [\![\tau]\!]^{\mathcal{I}_\mathrm{s} \cup \mathcal{I}_\mathrm{d}^\mathrm{r}} \to \mathbf{E}_\perp, \qquad \mathrm{reify}_\tau = \lambda a. \downarrow_\tau a\, (\eta^\perp\, 0)$$

Then for any closed term $\vdash E : \tau$, define the partial function,

$$\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E = \begin{cases} \tilde{E} & \textit{if}\ \mathrm{reify}_\tau\, ([\![E]\!]^{\mathcal{I}_\mathrm{s} \cup \mathcal{I}_\mathrm{d}^\mathrm{r}}\, \emptyset) = \eta^\perp \ulcorner \tilde{E} \urcorner \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

And again, we can state three properties of the normalization function:

Theorem 6 (Partial Correctness). *Let $\vdash_{\Sigma_\mathrm{s} \cup \Sigma_\mathrm{d}} E : \tau$ be a closed term of fully dynamic type. Then*

1. $\mathrm{norm}^{\mathcal{I}_\mathrm{s}}$ *is type-preserving (but not necessarily total): if* $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E = \tilde{E}$ *for some $\tilde{E}$ then $\vdash_{\Sigma_\mathrm{d}} \tilde{E} : \tau$ and $\vdash^{\mathrm{nf}} \tilde{E} : \tau$.*
2. *If* $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E = \tilde{E}$ *for some $\tilde{E}$ then $\mathcal{I}_\mathrm{s} \models \tilde{E} = E$.*
3. *If $\mathcal{I}_\mathrm{s} \models E = E'$ then* $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E$ *and* $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E'$ *are either both undefined or both defined and equal.*

The proof is somewhat more involved than in the pure lambda calculus case, since we can no longer exploit that any term over the full signature is statically equivalent to one in normal form (terms involving static fixed points may have no normal forms). We can, however, adapt an explicit normalization proof based on Kripke logical relations to the domain-theoretic case without too much additional trouble. The details can be found in [Fil99b], which also sketches how the normalization result relates to the correctness proof of the Lambda-mix partial evaluator [JGS93, Section 8.8].

Without strong normalization, another notion from rewriting enters: in general it may be that one series of reductions brings a term to normal form, while another gives rise to an infinite reduction sequence. We usually say that a reduction strategy is *complete* if it terminates with a normal form whenever the term can be reduced to normal form at all. For the pure, untyped lambda calculus, for example, one can show that always contracting the leftmost-outermost redex is a complete strategy for β-normalization. A similar property holds for our normalizer:

Theorem 7 (Completeness). *If for a term E, there exists an $\tilde{E}$ satisfying the conclusions of parts (1) and (2) of Theorem 6, then* $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E$ *is in fact defined.*

The proof proceeds by showing (through another simple logical-relations argument) that for any $\vdash_{\Sigma_\mathrm{d}} \tilde{E} : \tau$, $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} \tilde{E}$ is defined. Thus, if $\mathcal{I}_\mathrm{s} \models E = \tilde{E}$, then, by Property (3) of Theorem 6, $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E = \mathrm{norm}^{\mathcal{I}_\mathrm{s}} \tilde{E}$ and thus $\mathrm{norm}^{\mathcal{I}_\mathrm{s}} E$ must be defined. Again, the details can be found in [Fil99b].

4.4 A normalization algorithm

Note that, so far, we have only been considering a mathematical normalization function: from the denotation of a term in a particular domain-theoretic model, we can extract a syntactic representation of the normal form of the term, if it exists. We will now consider how to exploit this result to actually compute that normal form using a functional program.

The benefit of limiting the variation of interpretation to just base types and constants should now be apparent: instead of constructing an implementation of a non-standard denotational semantics, we can construct it in terms of an existing implementation of a standard semantics. We only need to construct a syntactic counterpart to the notion of a residualizing interpretation. This can be formalized as follows:

Definition 6. *A* realization Φ *of a signature* Σ *over a signature* Σ' *is a type-preserving substitution that assigns to every type constant of* Σ *a type phrase (not necessarily atomic) over* Σ'*, and to every term constant of* Σ *a term over* Σ'*. Applying such a substitution to a* Σ*-phrase (type or term)* θ*, we obtain a* Σ'*-phrase* $\theta\{\Phi\}$*.*

We assume now that we have a PCF-like programming language with signature $\Sigma_{\mathrm{pl}} \supseteq \Sigma_{\mathrm{s}}$ and interpretation $\mathcal{I}_{\mathrm{pl}}$ agreeing with $\mathcal{I}_{\mathrm{s}}$ on Σ_{s}, as well as an executable implementation of the corresponding evaluation function $\mathrm{eval}^{\mathcal{I}_{\mathrm{pl}}}$. For notational simplicity, we will also require that the programming language includes binary product types. (This is not essential, as we could have made all functions curried.)

Assume further that our programming language includes base types var and exp, with $\mathcal{B}_{\mathrm{pl}}(\mathsf{var}) = \mathbf{V}$ and $\mathcal{B}_{\mathrm{pl}}(\mathsf{exp}) = \mathbf{E}$. Moreover, let Σ_{pl} contain constructor constants corresponding to the semantic constructor functions, for example,

$$\Sigma_{\mathrm{pl}}(\mathsf{VAR}) \;=\; \mathsf{var} \to \mathsf{exp} \qquad \mathcal{C}_{\mathrm{pl}}(\mathsf{VAR}) \;=\; \lambda d^{\mathbf{V}_\perp}.\, d \star^{\perp} \lambda v.\, \eta^{\perp}\,(\mathrm{VAR}\, v)$$

(Note that we extend the semantic constructor function from sets to flat domains to fit into the interpretation of types.) Finally, we assume a function constant $\mathsf{mkvar} : \mathsf{int} \to \mathsf{var}$ such that for all $i \in \mathbb{N}$, $\mathcal{C}_{\mathrm{pl}}(\mathsf{mkvar})\,(\eta^{\perp}\, i) = \eta^{\perp}\, g_i$.

For the type part of the residualizing interpretation, we first define the abbreviation

$$\mathit{expf} \equiv \mathsf{int} \to \mathsf{exp}$$

and then take, for all dynamic base types $\underline{b}$,

$$\Phi^{\mathrm{r}}(\underline{b}) = \mathit{expf}$$

This gives us exactly $[\![\underline{b}\{\Phi^{\mathrm{r}}\}]\!]^{\mathcal{I}_{\mathrm{pl}}} = \hat{\mathbf{E}} = [\![\underline{b}]\!]^{\mathcal{I}_{\mathrm{s}} \cup \mathcal{I}^{\mathrm{r}}_{\mathrm{d}}}$, and thus $[\![\tau\{\Phi^{\mathrm{r}}\}]\!]^{\mathcal{I}_{\mathrm{pl}}} = [\![\tau]\!]^{\mathcal{I}_{\mathrm{s}} \cup \mathcal{I}^{\mathrm{r}}_{\mathrm{d}}}$ for all τ.

The wrapper functions are likewise denotable by Σ_{pl}-terms, for example,

$$\mathit{LAMF} : (\mathsf{var} \to \mathit{expf}) \to \mathit{expf}, \qquad \mathit{LAMF} \equiv \lambda\!\!\lambda f.\, \lambda\!\!\lambda i.\, \mathsf{LAM}\cdot(\mathsf{mkvar}\cdot i, f\cdot(i+1))$$

with $[\![\mathit{LAMF}]\!]^{\mathcal{I}_{\mathrm{pl}}} = \widehat{\mathrm{LAM}}$, and analogously for the other wrappers. Note that the explicit forcing in the semantic terms comes for free from the strict behavior of the term-constructor and arithmetic constants. We can then express the realizations of reflection and reification analogously to their semantic definitions:

$$\begin{aligned}
\mathit{reifyf}_\tau &: \tau\{\Phi^{\mathrm{r}}\} \dashrightarrow \mathit{expf} \\
\mathit{reifyf}_{\underline{b}} &\equiv \lambda e.\, e \\
\mathit{reifyf}_{\tau_1 \dashrightarrow \tau_2} &\equiv \lambda f.\, \mathit{LAMF}\cdot(\lambda v.\, \mathit{reifyf}_{\tau_2}\cdot(f\cdot(\mathit{reflectf}_{\tau_1}\cdot(\mathit{VARF}\cdot v)))) \\
\mathit{reflectf}_\tau &: \mathit{expf} \dashrightarrow \tau\{\Phi^{\mathrm{r}}\} \\
\mathit{reflectf}_{\underline{b}} &\equiv \lambda e.\, e \\
\mathit{reflectf}_{\tau_1 \dashrightarrow \tau_2} &\equiv \lambda e.\, \lambda a.\, \mathit{reflectf}_{\tau_2}\cdot(\mathit{APPF}\cdot(e, \mathit{reifyf}_{\tau_1}\cdot a))
\end{aligned}$$

with $[\![\mathit{reifyf}_\tau]\!]^{\mathcal{I}_{\mathrm{pl}}}\,\emptyset = \downarrow_\tau$ and $[\![\mathit{reflectf}_\tau]\!]^{\mathcal{I}_{\mathrm{pl}}}\,\emptyset = \uparrow_\tau$. And as the residualizing realization of the term constants in Σ_{d}, we finally take,

$$\begin{aligned}
\Phi^{\mathrm{r}}(\underline{c}_{\tau_1,\ldots,\tau_n}) &= \mathit{reflectf}_{\Sigma_{\mathrm{d}}(\underline{c}_{\tau_1,\ldots,\tau_n})}\cdot(\mathit{CSTF}\cdot c) \\
\Phi^{\mathrm{r}}(\$_b) &= \mathit{LITF}_b
\end{aligned}$$

It is now straightforward to show:

Theorem 8 (Implementing the CBN normalizer). *For any dynamic type τ, we define the term*

$$\mathit{reify}_\tau : \tau\{\Phi^{\mathrm{r}}\} \dashrightarrow \mathsf{exp}, \qquad \mathit{reify}_\tau \equiv \lambda a.\, \mathit{reifyf}_\tau\cdot a\cdot 0$$

Then the static normal form function of any closed term $\vdash_{\Sigma_{\mathrm{s}}\cup\Sigma_{\mathrm{d}}} E : \tau$ can be computed as:

$$\mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}\, E = \tilde{E} \quad \textit{iff} \quad \mathrm{eval}^{\mathcal{I}_{\mathrm{pl}}}(\mathit{reify}_\tau\cdot E\{\Phi^{\mathrm{r}}\}) = \ulcorner\tilde{E}\urcorner$$

In particular, for partial evaluation, if we have a function of two arguments, where the second argument and the result type are classified as dynamic, that is,

$$\vdash_{\Sigma_{\mathrm{s}}\cup\Sigma_{\mathrm{d}}} F : \sigma \dashrightarrow \tau \dashrightarrow \tau'$$

then for any static value $\vdash_{\Sigma_{\mathrm{s}}\cup\Sigma_{\mathrm{d}}} s : \sigma$, we can compute $\mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}(F\cdot s)$ to obtain the specialized program $\vdash_{\Sigma_{\mathrm{d}}} \widetilde{F_s} : \tau \dashrightarrow \tau'$.

Remark 3. The semantics of the type exp is actually a bit subtle. Note that our analysis assumes that $\mathcal{B}_{\mathrm{pl}}(\mathsf{exp})$ is a flat domain, with strict constructor functions. This is trivially satisfied if we take exp as, for example, the type of strings. However, if we simply use a Haskell-style datatype to define exp, it will also include many "partially defined" values because of the extra liftings of sum types. Worse, the constructor functions will not be strict, and the reification function may in fact produce partially defined results when static subcomputations diverge. Only when the normalized term has been completely printed can we be sure that the normalization function was in fact defined.

Remark 4. When there are no dynamic constants in the signature, the substitution Φ^{r} simply replaces all occurrences of $\underline{b}$ in type tags with exp. In a polymorphic language such as Haskell or ML, this allows a shortcut: one can simply leave all dynamic types in E as uninstantiated polymorphic type variables; the application of *reify* will instantiate them to exp. Moreover, any (monomorphic) dynamic constants can also be handled using explicit lambda-abstractions. This approach was used in early presentations of TDPE [Dan96], but gets awkward for larger examples. The functor-based approach described below scales up more gracefully.

Example 2. Returning to the possible annotations of the power function from Example 1, we get

$$\begin{aligned} \mathrm{norm}\,(\$\cdot(\mathit{power}_{ss}\cdot 3\cdot 4)) &= \$\cdot 81 \\ \mathrm{norm}\,(\dot{\lambda} x^{\underline{\mathsf{int}}}.\,\mathit{power}_{ds}\cdot x\cdot 3) &= \lambda g_0^{\underline{\mathsf{int}}}.\,g_0 \mathbin{\underline{\times}} (g_0 \mathbin{\underline{\times}} (g_0 \mathbin{\underline{\times}} \$\cdot 1)) \\ \mathrm{norm}\,(\dot{\lambda} x^{\underline{\mathsf{int}}}.\,\mathit{power}_{ds}\cdot x\cdot {-2}) &\quad \text{undefined} \end{aligned}$$

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.

As a further refinement, 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, applying the relevant substitutions does not even require an explicit syntactic traversal of the program, making it possible to enrich the static fragment of the language (for example, with pattern matching) without any modification to the partial evaluator itself.

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

$$\mathit{rr}(\alpha) \equiv (\alpha \dashrightarrow \mathit{expf}) \times (\mathit{expf} \dashrightarrow \alpha)\,.$$

Then for any dynamic type τ, we can construct a term of type $\mathit{rr}(\tau\{\Phi^{\mathrm{r}}\})$ whose value is the pair $(\mathit{reifyf}_\tau, \mathit{reflectf}_\tau)$. We do this by defining once and for all two ML-typable terms

$$\mathit{base} : \mathit{rr}(\mathit{expf}) \qquad \mathit{arrow} : \forall \alpha, \beta.\, \mathit{rr}(\alpha) \times \mathit{rr}(\beta) \dashrightarrow \mathit{rr}(\alpha \dashrightarrow \beta)\,,$$

with which we can systematically construct the required value. The technique is explained in more detail elsewhere [Yan98].

Finally, the dynamic polymorphic constants (for example, $\underline{\mathsf{fix}}$) 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, since the standard interpretations of dynamic constants are parametrically polymorphic; but the residualizing realization uses the type representations to construct the reify-reflect pair for $\Sigma(\underline{c}_{\tau_1,\ldots,\tau_n})$ given corresponding pairs for $\tau_1, \ldots, \tau_n$.

We do not show actual runnable code here, since there is no widely adopted CBN language with SML-style functors, allowing multiple implementations of a given signature to coexist in the same program. On the other hand, while the terms above can certainly be coded in ML, their behavior would be suspect for anything involving recursion. We refer the reader to Section 5.5 for actual SML code implementing normalization in a call-by-value language; the corresponding code for CBN is virtually identical, except for the actual definitions of the reification and reflection functions.

5 TDPE for call-by-value and computational effects

To complete the transition to practical programming, we need to account for the fact that execution of functional programs may involve general computational effects, both "internal" (such as catching and throwing exceptions) and "external" (such as performing I/O operations). It is well known that reasoning about such programs requires a more refined notion of equivalence than full $\beta\eta$-conversion.

We will look at computational effects in an ML-like call-by-value setting, where it will turn out that both the semantic and syntactic characterizations of normal forms differ significantly from the purely functional ones. We expect, however, that the results in this section can be adapted to programs in purely functional languages, written in "monadic style": we would then effectively be normalizing with respect to an equational theory including not only $\beta\eta$-conversion, but also the three monad laws.

The effectful setting is where the semantic notion of convertibility really shines: being able to reason about interpreted constants in terms of their denotations, rather than their operational behavior, proves to be a substantial help when proving the correctness of the final normalization-by-evaluation algorithm, since that algorithm itself is implemented in terms of computational effects.

5.1 A call-by-value language framework

As before, a program is a term over a signature of type and term constants. This time, however, the basic language is a bit richer. We add three new components:

- A let-expression for explicitly expressing sequencing.
- Binary products. (This is mainly for cosmetic reasons, as curried arithmetic primitives get somewhat ugly in CBV.)
- A primitive type of booleans and an if-expression. We could easily generalize to general disjoint-union types (see Exercise 8), but simple booleans illustrate the basic principles.

The set of types over Σ is therefore now:

$$\frac{b \in \Sigma}{\vdash_\Sigma b \ \mathit{type}} \qquad \frac{\vdash_\Sigma \sigma_1 \ \mathit{type} \quad \vdash_\Sigma \sigma_2 \ \mathit{type}}{\vdash_\Sigma \sigma_1 \rightharpoonup \sigma_2 \ \mathit{type}}$$

$$\frac{\vdash_\Sigma \sigma_1 \ \mathit{type} \quad \vdash_\Sigma \sigma_2 \ \mathit{type}}{\vdash_\Sigma \sigma_1 \times \sigma_2 \ \mathit{type}} \qquad \frac{}{\vdash_\Sigma \mathbf{bool} \ \mathit{type}}$$

The terms are much as before, with the straightforward addition of the following constructors:

$$\frac{\Gamma \vdash_\Sigma E_1 : \sigma_1 \quad \Gamma, x{:}\,\sigma_1 \vdash_\Sigma E_2 : \sigma_2}{\Gamma \vdash_\Sigma \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 : \sigma_2} \qquad \frac{\Gamma \vdash_\Sigma E_1 : \sigma_1 \quad \Gamma \vdash_\Sigma E_2 : \sigma_2}{\Gamma \vdash_\Sigma (E_1, E_2) : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash_\Sigma E : \sigma_1 \times \sigma_2 \quad \Gamma, x_1{:}\,\sigma_1, x_2{:}\,\sigma_2 \vdash_\Sigma E' : \sigma}{\Gamma \vdash_\Sigma \mathbf{match}\ (x_1, x_2) = E\ \mathbf{in}\ E' : \sigma}$$

$$\frac{}{\Gamma \vdash_\Sigma \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash_\Sigma \mathbf{false} : \mathbf{bool}}$$

$$\frac{\Gamma \vdash_\Sigma E_1 : \mathbf{bool} \quad \Gamma \vdash_\Sigma E_2 : \sigma \quad \Gamma \vdash_\Sigma E_3 : \sigma}{\Gamma \vdash_\Sigma \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 : \sigma}$$

(We use a pattern-matching construct instead of explicit projections, which could be defined like $\mathbf{fst}\,E \equiv \mathbf{match}\ (x,y) = E\ \mathbf{in}\ x$. This gives a more uniform treatment of normal forms, but is not essential. In practice, one would typically extend the language to allow general pattern matching in lambda- and let-bindings. See Exercise 9.)

Again, we say that a complete program is a closed term of base type.

For the semantics, we get a new component: an interpretation is now a quadruple $\mathcal{I} = (\mathcal{B}, \mathcal{L}, \mathcal{C}, \mathcal{T})$, where $\mathcal{T} = (T, \eta, \star)$ is a *monad* modeling the spectrum of effects in the language. To model recursion we require that $\mathcal{T}$ admits a monad morphism from the lifting monad, ensuring that any $A_\perp$-computation can be meaningfully seen as a TA-computation. Technically, the condition amounts to requiring that the cpo TA is pointed for any A, and that the function $\lambda t.\, t \star f \in TA \to TB$ is strict for any $f \in A \to TB$, that is, that $\perp_{TA} \star f = \perp_{TB}$.

(Of course, one can take the monad $\mathcal{T}$ to be simply lifting, but we will see in a moment why baking this choice into the semantic framework will prevent us from using NBE.)

We can then define the CBV semantics of types:

$$\begin{aligned} [\![b]\!]^{\mathcal{I}}_{\mathrm{v}} &= \mathcal{B}(b) \\ [\![\mathbf{bool}]\!]^{\mathcal{I}}_{\mathrm{v}} &= \mathbb{B} \\ [\![\sigma_1 \times \sigma_2]\!]^{\mathcal{I}}_{\mathrm{v}} &= [\![\sigma_1]\!]^{\mathcal{I}}_{\mathrm{v}} \times [\![\sigma_2]\!]^{\mathcal{I}}_{\mathrm{v}} \\ [\![\sigma_1 \to \sigma_2]\!]^{\mathcal{I}}_{\mathrm{v}} &= [\![\sigma_1]\!]^{\mathcal{I}}_{\mathrm{v}} \to T[\![\sigma_2]\!]^{\mathcal{I}}_{\mathrm{v}} \end{aligned}$$

Note that the meaning of a type is now a cpo that is not necessarily pointed. The meaning of a type assignment is still a product cpo,

$$[\![\Gamma]\!]^{\mathcal{I}}_{\mathrm{v}} = \prod_{x \in \mathrm{dom}\,\Gamma} [\![\Gamma(x)]\!]^{\mathcal{I}}_{\mathrm{v}}$$

but it will not in general be pointed, either.

The meaning of a term $\Gamma \vdash_\Sigma E : \sigma$ is now a continuous function $[\![E]\!]^{\mathcal{I}}_{\mathrm{v}} \in [\![\Gamma]\!]^{\mathcal{I}}_{\mathrm{v}} \to T[\![\sigma]\!]^{\mathcal{I}}_{\mathrm{v}}$,

$$\begin{array}{rcl}
[\![l]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& \eta\,(\mathcal{L}_b(l)) \\
[\![c_{\sigma_1,\ldots,\sigma_n}]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& \eta\,(\mathcal{C}(c_{\sigma_1,\ldots,\sigma_n})) \\
[\![\mathbf{true}]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& \eta\,\mathrm{true} \\
[\![\mathbf{false}]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& \eta\,\mathrm{false} \\
[\![x]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& \eta\,(\rho\,x) \\
[\![(E_1,E_2)]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& [\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a_1.\,[\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a_2.\,\eta\,\langle a_1,a_2\rangle \\
[\![\mathbf{match}\ (x_1,x_2)=E\ \mathbf{in}\ E']\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& [\![E]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda\langle a_1,a_2\rangle.\,[\![E']\!]^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x_1\mapsto a_1, x_2\mapsto a_2]) \\
[\![\lambda x^\sigma.\,E]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& \eta\,(\lambda a^{[\![\sigma]\!]^{\mathcal{I}}_{\mathrm{v}}}.\,[\![E]\!]^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x\mapsto a])) \\
[\![E_1\cdot E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& [\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda f.\,[\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a.\,f\,a \\
[\![\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& [\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda b.\,\mathrm{if}\ b\ \mathrm{then}\ [\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho\ \mathrm{else}\ [\![E_3]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \\
[\![\mathbf{let}\ x=E_1\ \mathbf{in}\ E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho &=& [\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \star \lambda a.\,[\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,(\rho[x\mapsto a])
\end{array}$$

(Note that the let-construct appears redundant, because we have

$$[\![\mathbf{let}\ x=E_1\ \mathbf{in}\ E_2]\!]^{\mathcal{I}}_{\mathrm{v}} = [\![(\lambda x.\,E_2)\,E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\ ,$$

but it turns out that including it gives a nicer syntactic characterization of normal forms.)

The standard static signature is similar to that of Definition 3, except that we no longer include the type constant bool, the associated literals true and false, or the constant if in Σ_{s}: those are now part of the fixed language core. Also, since it only makes sense to define recursive values of functional type, the fixed-point constants are now parameterized by two types,

$$\mathsf{fix}_{\sigma_1,\sigma_2} : ((\sigma_1 \to \sigma_2) \to \sigma_1 \to \sigma_2) \to \sigma_1 \to \sigma_2$$

Finally, we make the arithmetic and comparison operators uncurried, so that the infix operation $x+y$ now stands for $+\cdot(x,y)$, etc.

The standard interpretation of the constants is also what could be expected:

$$\begin{array}{rcl}
\mathcal{B}_{\mathrm{s}}(\mathsf{int}) &=& \mathbb{Z} \\
\mathcal{C}_{\mathrm{s}}(\diamond) &=& \lambda(x,y)^{\mathbb{Z}\times\mathbb{Z}}.\,\eta\,(x\diamond y) \quad {\scriptstyle \diamond\in\{+,-,\times,=,<\}} \\
\mathcal{C}_{\mathrm{s}}(\mathsf{fix}_{\sigma_1,\sigma_2}) &=& \lambda f^{([\![\sigma_1]\!]\to T[\![\sigma_2]\!])\to T([\![\sigma_1]\!]\to T[\![\sigma_2]\!])}. \\
&& \quad \eta\,(\bigsqcup_{i\in\omega}(\lambda g^{[\![\sigma_1]\!]\to T[\![\sigma_2]\!]}.\,\lambda a^{[\![\sigma_1]\!]}.\,f\,g \star \lambda g'.\,g'\,a)^i\,(\lambda a^{[\![\sigma_1]\!]}.\,\bot_{T[\![\sigma_2]\!]}))
\end{array}$$

Note how we make crucial use of the requirement that $T[\![\sigma_2]\!]$ must have a least element, for the interpretation of fix.

5.2 Binding times and static normalization

As before, we consider a binding-time annotated program to be expressed over a signature which has been explicitly partitioned into a static and a dynamic part. We still say that a general type is fully dynamic if it is constructed using base types from only the dynamic part of the signature.

For the interpretation of a partitioned signature, we require that all static constants have meanings *parametric* in the choice of monad, being able to rely only on TA being pointed, but not on any other structure of $\mathcal{T}$; this still allows us to include operations such as the fixed-point operator above, and possibly additional primitive partial functions. On the other hand, the meanings of the dynamic constants may be expressed with respect to a *specific* monad. Thus, in a combined interpretation $\mathcal{I}_s \cup \mathcal{I}_d$, all the constants can still be given a consistent interpretation based on the monad of $\mathcal{I}_d$. As before, we can then define a notion of static equivalence: $\mathcal{I}_s \models_v E = E'$ iff for all $\mathcal{I}_d$ interpreting the dynamic types, constants, and monad, $[\![E]\!]_v^{\mathcal{I}_s \cup \mathcal{I}_d} = [\![E']\!]_v^{\mathcal{I}_s \cup \mathcal{I}_d}$.

This notion of static equivalence captures transformations that are safe for any CBV language with monadic effects. For example, when x does not occur free in E_3, we always have

$$\mathcal{I}_s \models_v (\mathbf{let}\ y = (\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2)\ \mathbf{in}\ E_3) = (\mathbf{let}\ x = E_1\ \mathbf{in}\ \mathbf{let}\ y = E_2\ \mathbf{in}\ E_3)$$

Remark 5. Another interesting static equivalence is the following:

$$\mathcal{I}_s \models_v f \cdot (\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3) = \mathbf{if}\ E_1\ \mathbf{then}\ f \cdot E_2\ \mathbf{else}\ f \cdot E_3$$

This static equivalence is particularly remarkable since it does *not* hold in the CBN semantics, unless f is known to denote a strict function, or E_1 is guaranteed to converge. This equation, sometimes known as a commuting conversion, allows us to eliminate all **bool**-typed variables as soon as they are introduced (by a lambda-, match- or let-binding), which enables a simple CBV NBE result for booleans and sums. An NBE result for sum types in a CBN-like setting was obtained very recently [ADHS01], but the details are considerably more involved than for the CBV case treated below.

It is easy to show that the computational lambda calculus [Mog89] is sound with respect to static equivalence. (Under certain circumstances, one can also show that it is complete, much like $\beta\eta$-conversion was complete with respect to CBN static equivalence.)

For a syntactic characterization of normal forms, somewhat analogous to the CBN case, we now have notions of *normal values* and *normal computations*:

$$\frac{\Delta(x) = \underline{b}}{\Delta \vdash^{\mathrm{nv}} x : \underline{b}} \qquad \frac{l \in \Xi(b)}{\Delta \vdash^{\mathrm{nv}} \$ \cdot l : \underline{b}} \qquad \frac{\Delta \vdash^{\mathrm{nv}} E_1 : \tau_1 \qquad \Delta \vdash^{\mathrm{nv}} E_2 : \tau_2}{\Delta \vdash^{\mathrm{nv}} (E_1, E_2) : \tau_1 \times \tau_2}$$

$$\frac{}{\Delta \vdash^{\mathrm{nv}} \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\Delta \vdash^{\mathrm{nv}} \mathbf{false} : \mathbf{bool}} \qquad \frac{\Delta, x : \tau_1 \vdash^{\mathrm{nc}} E : \tau_2}{\Delta \vdash^{\mathrm{nv}} \underline{\lambda} x^{\tau_1}. E : \tau_1 \rightharpoonup \tau_2}$$

$$\frac{\Delta \vdash^{\mathrm{nv}} E : \tau}{\Delta \vdash^{\mathrm{nc}} E : \tau} \qquad \frac{\Delta(x) = \mathbf{bool} \qquad \Delta \vdash^{\mathrm{nc}} E_1 : \tau \qquad \Delta \vdash^{\mathrm{nc}} E_2 : \tau}{\Delta \vdash^{\mathrm{nc}} \mathbf{if}\ x\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 : \tau}$$

$$\frac{\Delta(x) = \tau_1 \times \tau_2 \qquad \Delta, x_1 : \tau_1, x_2 : \tau_2 \vdash^{\mathrm{nc}} E : \tau}{\Delta \vdash^{\mathrm{nc}} \textbf{match}\ (x_1, x_2) = x\ \textbf{in}\ E : \tau}$$

$$\frac{\Delta(x_2) = \tau_1 \dashrightarrow \tau_2 \qquad \Delta \vdash^{\mathrm{nv}} E_1 : \tau_1 \qquad \Delta, x_1 : \tau_2 \vdash^{\mathrm{nc}} E_2 : \tau}{\Delta \vdash^{\mathrm{nc}} \textbf{let}\ x_1 = x_2 \cdot E_1\ \textbf{in}\ E_2 : \tau}$$

$$\frac{\Sigma(c) = \tau_1 \dashrightarrow \tau_2 \qquad \Delta \vdash^{\mathrm{nv}} E_1 : \tau_1 \qquad \Delta, x : \tau_2 \vdash^{\mathrm{nc}} E_2 : \tau}{\Delta \vdash^{\mathrm{nc}} \textbf{let}\ x = c \cdot E_1\ \textbf{in}\ E_2 : \tau}$$

In particular, for terms not involving boolean or product types, a normal value is either a base-typed constant or variable, or of the form

$$\lambda\!\!\!^{-} x.\,\textbf{let}\ x_1 = f_1 \cdot V_1\ \textbf{in}\ \cdots\ \textbf{let}\ x_n = f_n \cdot V_n\ \textbf{in}\ V_{n+1} \quad (n \geq 0)\ ,$$

where all the V_i are normal values, and each f_i is a function-typed constant or variable.

(These rules are actually a bit too permissive, in that they admit both variants of statically equivalent terms, such as

$$\lambda\!\!\!^{-} x^{\mathbf{bool}}.\,\textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ 4 \quad \text{and} \quad \lambda\!\!\!^{-} x^{\mathbf{bool}}.\,\textbf{if}\ x\ \textbf{then}\ 3\ \textbf{else}\ \textbf{if}\ x\ \textbf{then}\ 5\ \textbf{else}\ 4$$

Of course, a proper normalization function can only return one of those. To get a more precise syntactic characterization of normal forms, one needs to further restrict the occurrences of sum- and product-typed variables, and to pick a canonical order for their elimination. This can be done fairly straightforwardly using a split typing context [Fil01]; we omit the details here.)

Remark 6. The observant reader may note that the rules for normal forms essentially correspond (modulo implicit uses of contraction and weakening) to the left- and right-introduction rules for a Gentzen-style intuitionistic sequent calculus. (The general form of the if-rule as a case expression introduces bound variables corresponding to the disjuncts.) Moreover, the typing of the general let corresponds to a cut. In other words, reducing CBV terms to normal form is closely related to cut-elimination. On the other hand, for the CBN normalizer, the rules correspond to the usual introduction and elimination rules in a natural-deduction calculus. It would be interesting to further investigate these connections to standard proof-normalization theory. Some results in this direction have already been obtained by Ohori [Oho99]; see [Fil01] for a more detailed discussion of this issue.

5.3 A residualizing interpretation for CBV

We now aim to find an NBE result for the CBV setting. Note that for the residualizing interpretation, we need a more "powerful" effect than simple partiality (as embodied by the lifting monad), because we need to distinguish between terms such as

$$E_1 \equiv \lambda\!\!\!^{-} f.\,\lambda\!\!\!^{-} x.\,(\lambda\!\!\!^{-} y.\,x) \cdot (f \cdot x) \quad \text{and} \quad E_2 \equiv \lambda\!\!\!^{-} f.\,\lambda\!\!\!^{-} x.\,x$$

which have observably different behavior (even when the effect is only partiality), and therefore should have different normal forms. But taking partiality as the residualizing effect will not allow us to extract enough information from the residualizing interpretations of those two terms to reconstruct them accurately: for any $\mathcal{I}$ whose $\mathcal{T}$ is the lifting monad, we have $[\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho \sqsubseteq [\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\rho$ (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 reify $\in [\![(b \rightharpoonup b) \rightharpoonup b \rightharpoonup b]\!]^{\mathcal{I}}_{\mathrm{v}} \to \mathbf{E}_\bot$ such that $\mathrm{reify}([\![E_1]\!]^{\mathcal{I}}_{\mathrm{v}}\,\emptyset) = \eta^\bot\,\ulcorner E_1 \urcorner$ and $\mathrm{reify}([\![E_2]\!]^{\mathcal{I}}_{\mathrm{v}}\,\emptyset) = \eta^\bot\,\ulcorner E_2 \urcorner$.

Instead, we pick for the residualizing interpretation a "universal" effect, which will ensure that we can probe the residualizing semantic interpretations closely enough to make all the required distinctions.

Incidentally, such a choice also allows the task of fresh-name generation to be folded into the residualization monad, instead of requiring us to work with term families such as $\hat{\mathbf{E}}$. We can now use an effect-based notion of freshness, generating "globally unique" variable names; with a suitable effect structure, we can actually make this concept precise.

We first define the name-generation monad. This is just a state-passing monad on top of partiality; the state is the "next free index":

$$\begin{aligned} T^{\mathrm{g}} A &= \mathbb{Z} \rightharpoonup (A \times \mathbb{Z})_\bot \\ \eta^{\mathrm{g}}\, a &= \lambda i.\, \eta^\bot \langle a, i\rangle \\ t \star^{\mathrm{g}} f &= \lambda i.\, t\, i \star^\bot \lambda\langle a, i'\rangle.\, f\, a\, i' \end{aligned}$$

(As in CBN, we use integers rather than natural numbers for the index, in anticipation of embedding the construction into an existing programming language.) Using T^{g} we can define an effectful computation that generates a fresh name, and one that initializes the counter for a delimited subcomputation:

$$\begin{aligned} \mathrm{new} &\in T^{\mathrm{g}}\mathbf{V} & \mathrm{withct}_A &\in T^{\mathrm{g}} A \to T^{\mathrm{g}} A \\ \mathrm{new} &= \lambda i.\, \eta^\bot \langle g_i, i+1\rangle & \mathrm{withct}_A &= \lambda t.\, \lambda i.\, t\, 0 \star^\bot \lambda\langle a, i'\rangle.\, \eta^\bot \langle a, i\rangle \end{aligned}$$

Further, we define T^{c} to be the continuation monad with answer domain chosen as $T^{\mathrm{g}}\mathbf{E}$:

$$\begin{aligned} T^{\mathrm{c}} A &= (A \rightharpoonup T^{\mathrm{g}}\mathbf{E}) \rightharpoonup T^{\mathrm{g}}\mathbf{E} \\ \eta^{\mathrm{c}}\, a &= \lambda\kappa.\, \kappa\, a \\ t \star^{\mathrm{c}} f &= \lambda\kappa.\, t\,(\lambda a.\, f\, a\, \kappa) \end{aligned}$$

Every T^{g}-computation can be seen as a T^{c}-computation without control effects, through the monad morphism $\gamma^{\mathrm{g,c}}_A \in T^{\mathrm{g}} A \to T^{\mathrm{c}} A$ given by

$$\gamma^{\mathrm{g,c}}_A\, t = \lambda\kappa^{A \rightharpoonup T^{\mathrm{g}}\mathbf{E}}.\, t \star^{\mathrm{g}} \kappa.$$

In particular, we can "lift" the name-generation functions to T^{c}-computations as

$$\begin{aligned} \mathrm{cnew} &\in T^{\mathrm{c}}\mathbf{V} & \mathrm{cwithct}_A &\in T^{\mathrm{c}} A \to T^{\mathrm{c}} A \\ \mathrm{cnew} &= \lambda\kappa.\, \lambda i.\, \kappa\, g_i\,(i+1) & \mathrm{cwithct}_A\, t &= \lambda\kappa.\, \lambda i.\, t\,(\lambda a.\, \lambda i'.\, \kappa\, a\, i)\, 0 \end{aligned}$$

These satisfy the natural equations $\mathrm{cnew} = \gamma^{\mathrm{g,c}}\mathrm{new}$ and $\mathrm{cwithct}\,(\gamma^{\mathrm{g,c}} t) = \gamma^{\mathrm{g,c}}(\mathrm{withct}\, t)$.

In any continuation monad, we can define operators for capturing complete continuations (such as Scheme's `call/cc`). However, for our particular choice of answer domain, we can also define operations for working with *delimited* or *composable* continuations [DF90]:

$$\begin{aligned}
\mathrm{reset} &\in T^{\mathrm{c}}\mathbf{E} \to T^{\mathrm{c}}\mathbf{E} \\
\mathrm{reset}\, t &= \gamma^{\mathrm{g,c}}_{\mathbf{E}}(t\,\eta^{\mathrm{g}}) = \lambda\kappa.\, t\,\eta^{\mathrm{g}} \star^{\mathrm{g}} \kappa \\
\mathrm{shift}_A &\in ((A \to T^{\mathrm{c}}\mathbf{E}) \to T^{\mathrm{c}}\mathbf{E}) \to T^{\mathrm{c}} A \\
\mathrm{shift}_A\, h &= \lambda\kappa.\, h\,(\lambda a.\, \gamma^{\mathrm{g,c}}_{\mathbf{E}}(\kappa\, a))\,\eta^{\mathrm{g}} = \lambda\kappa.\, h\,(\lambda a.\, \lambda\kappa'.\, \kappa\, a \star^{\mathrm{g}} \kappa')\,\eta^{\mathrm{g}}
\end{aligned}$$

Here, $\mathrm{reset}\, t$ evaluates t with an empty continuation, thus encapsulating any control effects t might have. $\mathrm{shift}\, h$ captures and removes the current continuation (up to the nearest enclosing reset), and passes it to h as a control-effect-free function.

These definitions are somewhat awkward to work with directly. However, we can easily check that they validate the following equational reasoning principles:

$$\begin{aligned}
\mathrm{reset}\,(\eta^{\mathrm{c}}\, a) &= \eta^{\mathrm{c}}\, a \\
\mathrm{reset}\,(\mathrm{shift}\, h \star^{\mathrm{c}} f) &= \mathrm{reset}\,(h\,(\lambda a.\, \mathrm{reset}\,(f\, a))) \\
\mathrm{reset}\,(\gamma^{\mathrm{g,c}} t \star^{\mathrm{c}} f) &= \gamma^{\mathrm{g,c}} t \star^{\mathrm{c}} \lambda a.\, \mathrm{reset}\,(f\, a)
\end{aligned}$$

The first of these says that a reset does not affect effect-free computations. The second specifies the behavior of a delimited shift-operation. (Remember that $\mathrm{shift}\, h = \mathrm{shift}\, h \star^{\mathrm{c}} \eta^{\mathrm{c}}$, and $(\mathrm{shift}\, h \star^{\mathrm{c}} f) \star^{\mathrm{c}} g = \mathrm{shift}\, h \star^{\mathrm{c}} (\lambda a.\, f\, a \star^{\mathrm{c}} g)$ by the usual monad laws, so the equation is widely applicable.) The third allows computations such as cnew (without control effects, but not necessarily completely effect-free) to be moved outside a reset. For example, we can derive

$$\begin{aligned}
&\mathrm{reset}\,\big(\mathrm{shift}\,(\lambda k.\, k\, 3 \star^{\mathrm{c}} \lambda x.\, \eta^{\mathrm{c}}(x+1)) \star^{\mathrm{c}} \lambda r.\, \eta^{\mathrm{c}}(2 \times r)\big) \star^{\mathrm{c}} \lambda a.\, \eta^{\mathrm{c}}(-a) \\
&\quad = \mathrm{reset}\,\big((\lambda k.\, k\, 3 \star^{\mathrm{c}} \lambda x.\, \eta^{\mathrm{c}}(x+1))\,(\lambda a.\, \mathrm{reset}\,(\eta^{\mathrm{c}}(2 \times a)))\big) \star^{\mathrm{c}} \lambda a.\, \eta^{\mathrm{c}}(-a) \\
&\quad = \mathrm{reset}\,\big(\mathrm{reset}\,(\eta^{\mathrm{c}}\, 6) \star^{\mathrm{c}} \lambda x.\, \eta^{\mathrm{c}}(x+1)\big) \star^{\mathrm{c}} \lambda a.\, \eta^{\mathrm{c}}(-a) \\
&\quad = \mathrm{reset}\,\big(\eta^{\mathrm{c}}\, 6 \star^{\mathrm{c}} \lambda x.\, \eta^{\mathrm{c}}(x+1)\big) \star^{\mathrm{c}} \lambda a.\, \eta^{\mathrm{c}}(-a) \\
&\quad = \mathrm{reset}\,(\eta^{\mathrm{c}}\, 7) \star^{\mathrm{c}} \lambda a.\, \eta^{\mathrm{c}}(-a) = \eta^{\mathrm{c}}(-7)
\end{aligned}$$

Note how the doubling operation (which uses the result of shift directly) gets captured as part of k, while the outer negation (which is protected by a reset) is not. It is this ability of shift to reschedule "future" computations that will allow us to properly arrange the residual code as it is being incrementally generated.

We can now take as the non-standard interpretation of dynamic base types and effects,

$$\mathcal{B}^{\mathrm{r}}_{\mathrm{d}}(\underline{b}) = \mathbf{E} \qquad \text{and} \qquad \mathcal{T}^{\mathrm{r}} = \mathcal{T}^{\mathrm{c}}$$

which again allows us to define reification and reflection functions:

$$\begin{array}{rcl}
\downarrow^{\mathrm{v}}_{\tau} & \in & [\![\tau]\!]^{\mathcal{I}_{\mathrm{s}}\cup\mathcal{I}^{\mathrm{r}}_{\mathrm{d}}}_{\mathrm{v}} \to T^{\mathrm{r}}\mathbf{E} \\
\downarrow^{\mathrm{v}}_{\underline{b}} & = & \lambda e.\,\eta^{\mathrm{r}}\,e \\
\downarrow^{\mathrm{v}}_{\mathbf{bool}} & = & \lambda b.\,\mathrm{if}\ b\ \mathrm{then}\ \eta^{\mathrm{r}}\,\mathrm{TRUE}\ \mathrm{else}\ \eta^{\mathrm{r}}\,\mathrm{FALSE} \\
\downarrow^{\mathrm{v}}_{\tau_1\times\tau_2} & = & \lambda\langle a_1,a_2\rangle.\,\downarrow^{\mathrm{v}}_{\tau_1} a_1 \star^{\mathrm{r}} \lambda e_1.\,\downarrow^{\mathrm{v}}_{\tau_2} a_2 \star^{\mathrm{r}} \lambda e_2.\,\eta^{\mathrm{r}}\,(\mathrm{PAIR}\,\langle e_1,e_2\rangle) \\
\downarrow^{\mathrm{v}}_{\tau_1\to\tau_2} & = &
\end{array}$$

$$\lambda f.\,\mathrm{cnew} \star^{\mathrm{r}} \lambda v.\,\mathrm{reset}\,\big(\uparrow^{\mathrm{v}}_{\tau_1}(\mathrm{VAR}\,v) \star^{\mathrm{r}} \lambda a.\,f\,a \star^{\mathrm{r}} \lambda b.\,\downarrow^{\mathrm{v}}_{\tau_2} b\big) \star^{\mathrm{r}} \lambda e.\,\eta^{\mathrm{r}}\,(\mathrm{LAM}\,\langle v,e\rangle)$$

$$\begin{array}{rcl}
\uparrow^{\mathrm{v}}_{\tau} & \in & \mathbf{E} \to T^{\mathrm{r}}[\![\tau]\!]^{\mathcal{I}_{\mathrm{s}}\cup\mathcal{I}^{\mathrm{r}}_{\mathrm{d}}}_{\mathrm{v}} \\
\uparrow^{\mathrm{v}}_{\underline{b}} & = & \lambda e.\,\eta^{\mathrm{r}}\,e \\
\uparrow^{\mathrm{v}}_{\mathbf{bool}} & = & \lambda e.\,\mathrm{shift}\,(\lambda k.\,k\,\mathrm{true} \star^{\mathrm{r}} \lambda e_1.\,k\,\mathrm{false} \star^{\mathrm{r}} \lambda e_2.\,\eta^{\mathrm{r}}\,(\mathrm{IF}\,\langle e,e_1,e_2\rangle)) \\
\uparrow^{\mathrm{v}}_{\tau_1\times\tau_2} & = &
\end{array}$$

$$\begin{array}{l}
\lambda e.\,\mathrm{shift}\,(\lambda k.\,\mathrm{cnew} \star^{\mathrm{r}} \lambda v_1.\,\mathrm{cnew} \star^{\mathrm{r}} \lambda v_2. \\
\qquad\qquad \mathrm{reset}\,\big(\uparrow^{\mathrm{v}}_{\tau_1}(\mathrm{VAR}\,v_1) \star^{\mathrm{r}} \lambda a_1.\,\uparrow^{\mathrm{v}}_{\tau_2}(\mathrm{VAR}\,v_2) \star^{\mathrm{r}} \lambda a_2.\,k\,\langle a_1,a_2\rangle\big) \star^{\mathrm{r}} \lambda e'. \\
\qquad\qquad\quad \eta^{\mathrm{r}}\,(\mathrm{MATCH}\,\langle\langle v_1,v_2\rangle,e,e'\rangle))
\end{array}$$

$$\begin{array}{rcl}
\uparrow^{\mathrm{v}}_{\tau_1\to\tau_2} & = & \lambda e.\,\eta^{\mathrm{r}}(\lambda a.\,\mathrm{cnew} \star^{\mathrm{r}} \lambda v. \\
 & & \qquad\quad \mathrm{shift}\,(\lambda k.\,\downarrow^{\mathrm{v}}_{\tau_1} a \star^{\mathrm{r}} \lambda e'.\,\mathrm{reset}\,(\uparrow^{\mathrm{v}}_{\tau_2}(\mathrm{VAR}\,v) \star^{\mathrm{r}} k) \star^{\mathrm{r}} \lambda e''. \\
 & & \qquad\qquad\quad \eta^{\mathrm{r}}\,(\mathrm{LET}\,\langle v,\mathrm{APP}\,\langle e,e'\rangle,e''\rangle)))
\end{array}$$

Observe how the non-trivial reflection functions use shift to wrap a syntactic binding or a test around their invocation points, which are expecting semantic results. Note also how reflection of a boolean expression sequentially traces through both possibilities for the boolean value returned, by passing both true and false to the continuation k.

Remark 7. It is evident that a new variable is generated every time the pseudo-semantic function returned by $\uparrow_{\tau_1\to\tau_2} e$ is applied. Had we instead, incorrectly, written $\uparrow_{\tau_1\to\tau_2}$ as $\lambda e.\,\mathrm{cnew} \star^{\mathrm{r}} \lambda v.\,\eta^{\mathrm{r}}\,(\lambda a.\,\mathrm{shift}\cdots)$, all such applications would share the same let-variable name, causing clashes. This illustrates how monads allow us to be precise about "freshness" in a way that informal annotations of definitions do not readily support.

The residualizing interpretation of dynamic constants and lifting is now:

$$\begin{array}{rcl}
\mathcal{C}^{\mathrm{r}}_{\mathrm{d}}(\underline{c}_{\tau_1,\ldots,\tau_n}) & = & \uparrow^{\mathrm{v}}_{\Sigma(c_{\tau_1,\ldots,\tau_n})}(\mathrm{CST}\,c) \\
\mathcal{C}^{\mathrm{r}}_{\mathrm{d}}(\$_b) & = & \lambda n^{\mathcal{B}_{\mathrm{s}}(b)}.\,\eta^{\mathrm{r}}\,(\mathrm{LIT}_b\,n)
\end{array}$$

Finally, we can again define a static-normalization function. For CBN, we could restrict ourselves to normalizing closed terms, because free variables could be lambda-abstracted without changing the result. For CBV, we make the additional restriction that the term to be normalized must be a formal value (constant, variable, lifted literal, or lambda-abstraction); we can always wrap a non-value term E in a dummy lambda-abstraction $\underline{\lambda} d.\,E$. We thus take:

Definition 7. *For any dynamic type τ, define the auxiliary function*

$$\mathrm{reify}^{\mathrm{v}}_{\tau} \in [\![\tau]\!]^{\mathcal{I}_{\mathrm{s}} \cup \mathcal{I}^{\mathrm{r}}_{\mathrm{d}}}_{\mathrm{v}} \to T^{\mathrm{r}}\mathbf{E}, \qquad \mathrm{reify}^{\mathrm{v}}_{\tau} = \lambda a.\, \mathrm{cwithct}_{\mathbf{E}}\, (\downarrow^{\mathrm{v}}_{\tau} a)$$

(Note how initialization of the counter now happens using computational effects.) Then, for any closed value $\vdash_{\Sigma_{\mathrm{s}} \cup \Sigma_{\mathrm{d}}} E : \tau$, we define the normalization function

$$\mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}_{\mathrm{v}}\, E = \begin{cases} \tilde{E} & \textit{if } \mathrm{reify}^{\mathrm{v}}_{\tau}\, ([\![E]\!]^{\mathcal{I}_{\mathrm{s}} \cup \mathcal{I}^{\mathrm{r}}_{\mathrm{d}}}_{\mathrm{v}}\, \emptyset) = \eta^{\mathrm{r}}\, \ulcorner \tilde{E} \urcorner \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

Again, we can summarize the result:

Theorem 9 (Correctness of CBV normalizer). *Partial correctness:*

1. *For any closed value $\vdash_{\Sigma_{\mathrm{s}} \cup \Sigma_{\mathrm{d}}} E : \tau$, if $\mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}_{\mathrm{v}}\, E = \tilde{E}$ for some $\tilde{E}$ then $\vdash^{\mathrm{nv}}_{\Sigma_{\mathrm{d}}} \tilde{E} : \tau$.*
2. *If $\mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}_{\mathrm{v}}\, E = \tilde{E}$ then $\mathcal{I}_{\mathrm{s}} \models_{\mathrm{v}} E = \tilde{E}$.*
3. *if $\mathcal{I}_{\mathrm{s}} \models_{\mathrm{v}} E = E'$ then $\mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}_{\mathrm{v}}\, E = \mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}_{\mathrm{v}}\, E'$ (both defined and equal, or both undefined).*

Completeness: If E is statically equivalent to some term in normal form, $\mathrm{norm}^{\mathcal{I}_{\mathrm{s}}}_{\mathrm{v}}\, E$ is defined.

The proofs for both halves are similar to the CBN case, but somewhat more elaborate, due to the extra parameterization on a dynamic monad, and the more complicated reification and reflection functions.

Remark 8. It is possible to define a CBV normalization function using only a suitable state monad in the residualizing interpretation, rather than continuations [SK01, Section 4]. This approach, however, does not allow the normalizer to handle booleans and sum types as part of the semantic framework.

5.4 A CBV normalization algorithm

Again, we can formulate the semantic normalization results in terms of a syntactic realization of the components. As before, we assume that our programming language has types var, exp, and constants corresponding to the constructor functions, for example,

$$\Sigma_{\mathrm{pl}}(\mathsf{APP}) = \mathsf{exp} \times \mathsf{exp} \rightarrow \mathsf{exp} \qquad \mathcal{C}_{\mathrm{pl}}(\mathsf{APP}) = \lambda\langle e_1, e_2\rangle.\, \eta^{\mathrm{r}}\, (\mathrm{APP}\, \langle e_1, e_2\rangle)$$

(Note that, unlike for the CBN case, the arguments of the constructor functions are values, not computations)

We also assume that $\mathcal{T}_{\mathrm{pl}} = \mathcal{T}^{\mathrm{r}}$, and that we have constants whose interpretations correspond to the semantic functions for name generation and control primitives:

$$\begin{aligned} \Sigma_{\mathrm{pl}}(\mathsf{gensym}) &= 1 \rightarrow \mathsf{var} & \mathcal{C}_{\mathrm{pl}}(\mathsf{gensym}) &= \lambda\langle\rangle.\, \mathrm{cnew} \\ \Sigma_{\mathrm{pl}}(\mathsf{withct}_{\sigma}) &= (1 \rightarrow \sigma) \rightarrow \sigma & \mathcal{C}_{\mathrm{pl}}(\mathsf{withct}_{\sigma}) &= \lambda t.\, \mathrm{cwithct}\, (t\, \langle\rangle) \\ \Sigma_{\mathrm{pl}}(\mathsf{reset}) &= (1 \rightarrow \mathsf{exp}) \rightarrow \mathsf{exp} & \mathcal{C}_{\mathrm{pl}}(\mathsf{reset}) &= \lambda t.\, \mathrm{reset}\, (t\, \langle\rangle) \\ \Sigma_{\mathrm{pl}}(\mathsf{shift}_{\sigma}) &= ((\sigma \rightarrow \mathsf{exp}) \rightarrow \mathsf{exp}) \rightarrow \sigma & \mathcal{C}_{\mathrm{pl}}(\mathsf{shift}_{\sigma}) &= \mathrm{shift}_{[\![\sigma]\!]} \end{aligned}$$

Like in the CBN case, a practical programming language usually has many further types and constants, beyond what we need to construct the residualizing interpretation of the dynamic signature. Additionally, a CBV language typically has a wider spectrum of effects than captured by the residualizing monad T^{r}. (For example, it may allow functions to have side effects or to perform I/O operations.) And it is not obvious that this additional generality of the programming language will not get in the way of our result.

What we require is that we can *simulate* the behavior of a T^{r}-computation using whatever more general monad the programming language provides. In fact, we only need to consider evaluation of programs without top-level residualization-specific effects. In other words, we only require that the evaluation partial function for complete programs (closed terms of base type) satisfies:

$$\mathrm{eval}_{\mathrm{v}}^{\mathcal{I}_{\mathrm{pl}}}\, E = \begin{cases} l & \text{if } [\![E]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{pl}}}\,\emptyset = \eta^{\mathrm{pl}}\,(\mathcal{L}_b\,(l)) \\ \text{undefined} & \text{if } [\![E]\!]_{\mathrm{v}}^{\mathcal{I}_{\mathrm{pl}}}\,\emptyset = \bot \end{cases}$$

Note that, unlike the case for CBN, these two possibilities are not exhaustive: the result of $\mathrm{eval}_{\mathrm{v}}^{\mathcal{I}_{\mathrm{pl}}}$ is unspecified for programs with top-level effects other than divergence, for example, those that try to capture the top-level continuation, or rely on the initial value of the gensym counter. Evaluation of such programs may abort, diverge, or even return unpredictable results. Of course, the program E that we evaluate to execute the normalization algorithm will not have such effects.

One can show that an $\mathrm{eval}_{\mathrm{v}}^{\mathcal{I}_{\mathrm{pl}}}$ with the above properties can be implemented through a further realization of Σ_{pl} in any general-purpose functional language that supports first-class continuations and references [Fil99a]. (In fact, we can implement a whole hierarchy of monads by such an embedding, and thus avoid the need to explicitly lift many monad operations, such as withct to cwithct.) The details are beyond the scope of these notes, but we do show the actual construction in Section 5.5.

We can now simply take the residualizing realizations of all dynamic base types as the type of term representations,

$$\Phi_{\mathrm{v}}^{\mathrm{r}}(\underline{b}) = \mathsf{exp}$$

and express the CBV reification and reflection functions as Σ_{pl}-terms with effects:

$$\begin{aligned}
\mathit{reifyve}_{\tau} \;&:\; \tau\{\Phi_{\mathrm{v}}^{\mathrm{r}}\} \rightharpoonup \mathsf{exp} \\
\mathit{reifyve}_{\underline{b}} \;&\equiv\; \lambda\!\!\!{}^{-} e.\, e \\
\mathit{reifyve}_{\mathbf{bool}} \;&\equiv\; \lambda\!\!\!{}^{-} b.\, \mathbf{if}\; b\; \mathbf{then}\; \mathsf{TRUE}\; \mathbf{else}\; \mathsf{FALSE} \\
\mathit{reifyve}_{\tau_1 \times \tau_2} \;&\equiv\; \lambda\!\!\!{}^{-} p.\, \mathbf{match}\; (x_1, x_2) = p\; \mathbf{in}\; \mathsf{PAIR}\cdot(\mathit{reifyve}_{\tau_1}\cdot a_1, \mathit{reifyve}_{\tau_2}\cdot a_2) \\
\mathit{reifyve}_{\tau_1 \rightharpoonup \tau_2} \;&\equiv\; \lambda\!\!\!{}^{-} f.\, \mathbf{let}\; v = \mathsf{gensym}\cdot() \\
&\qquad \mathbf{in}\; \mathsf{LAM}\cdot(v, \mathsf{reset}\cdot(\lambda\!\!\!{}^{-}().\, \mathit{reifyve}_{\tau_2}\cdot(f\cdot(\mathit{reflectve}_{\tau_1}\cdot(\mathsf{VAR}\cdot v)))))
\end{aligned}$$

$$\begin{array}{rcl}
\mathit{reflectve}_{\tau} & : & \mathsf{exp} \dot{\rightarrow} \tau\{\Phi^{\mathrm{r}}_{\mathrm{v}}\} \\
\mathit{reflectve}_{\underline{b}} & \equiv & \dot{\lambda} e.\, e \\
\mathit{reflectve}_{\mathbf{bool}} & \equiv & \dot{\lambda} e.\, \mathsf{shift}\cdot(\dot{\lambda} k.\, \mathsf{IF}\cdot(e, k\cdot\mathbf{true}, k\cdot\mathbf{false})) \\
\mathit{reflectve}_{\tau_1 \dot{\times} \tau_2} & \equiv & \\
\end{array}$$

$$\begin{array}{l}
\dot{\lambda} e.\, \mathbf{let}\ v_1 = \mathsf{gensym}\cdot() \\
\quad \mathbf{in\ let}\ v_2 = \mathsf{gensym}\cdot() \\
\qquad \mathbf{in}\ \mathsf{shift}\cdot(\dot{\lambda} k.\, \mathsf{MATCH}\cdot((v_1, v_2), e, \\
\qquad\qquad \mathsf{reset}\cdot(\dot{\lambda}().\, k\cdot(\mathit{reflectve}_{\tau_1}\cdot(\mathsf{VAR}\cdot v_1), \mathit{reflectve}_{\tau_2}\cdot(\mathsf{VAR}\cdot v_2)))))
\end{array}$$

$$\begin{array}{rcl}
\mathit{reflectve}_{\tau_1 \dot{\rightarrow} \tau_2} & \equiv & \dot{\lambda} e.\, \dot{\lambda} a.\, \mathbf{let}\ v = \mathsf{gensym}\cdot() \\
& & \qquad \mathbf{in}\ \mathsf{shift}\,(\lambda k.\, \mathsf{LET}\cdot(v, \mathsf{APP}\cdot(e, \mathit{reifyve}_{\tau_1}\cdot a), \\
& & \qquad\qquad \mathsf{reset}\cdot(\dot{\lambda}().\, k\cdot(\mathit{reflectve}_{\tau_2}\cdot(\mathsf{VAR}\cdot v)))))
\end{array}$$

Note that, although these terms are significantly more complicated than their CBN counterparts, they still share some basic structure. For reification at functional type, we still reflect a new VAR, apply the function, reify the result, and generate a LAM. Similarly, for reflection, we reify the function argument, generate an APP, and reflect the result (bound to an intermediate variable this time.) The main conceptual change is in the added shift and reset operations to suitably rearrange the generated code.

We complete the residualizing realization of the dynamic signature by taking

$$\begin{array}{rcl}
\Phi^{\mathrm{r}}_{\mathrm{v}}(\underline{c}_{\tau_1,\ldots,\tau_n}) & = & \mathit{reflectve}_{\Sigma(\underline{c}_{\tau_1,\ldots,\tau_n})}\cdot(\mathsf{CST}\cdot c) \\
\Phi^{\mathrm{r}}_{\mathrm{v}}(\$_b) & = & \dot{\lambda} n.\, \mathsf{LIT}_b\cdot n
\end{array}$$

Finally, we can state again a procedure for computing CBV normal forms:

Theorem 10 (Implementing the CBV normalizer). *We define the auxiliary term*

$$\mathit{reifyv}_{\tau} : \tau\{\Phi^{\mathrm{r}}_{\mathrm{v}}\} \dot{\rightarrow} \mathsf{exp}, \qquad \mathit{reifyv}_{\tau} \equiv \dot{\lambda} a.\, \mathsf{withct}_{\mathsf{exp}}\cdot(\dot{\lambda}().\, \mathit{reifyve}_{\tau}\cdot a)$$

Then for any closed value $\vdash_{\Sigma_s \cup \Sigma_d} E : \tau$*, its CBV static normal form can be computed as*

$$\mathrm{norm}^{\mathcal{I}_s}_{\mathrm{v}}\, E = \tilde{E} \ \ \textit{iff} \ \ \mathrm{eval}^{\mathcal{I}_{\mathrm{pl}}}_{\mathrm{v}}\,(\mathit{reifyv}_{\tau}\cdot E\{\Phi^{\mathrm{r}}_{\mathrm{v}}\}) = \ulcorner \tilde{E} \urcorner$$

For partial evaluation, we again take the E to be normalized as the partial application of the binding-time separated original program to the static argument:

Example 3. Let us revisit the power function from Example 1 in a CBV setting. With conditionals now being part of the framework, here are the two interesting annotations of power:

$$\begin{array}{l}
\mathit{power}_{ds} : \underline{\iota} \dot{\rightarrow} \overline{\iota} \dot{\rightarrow} \underline{\iota} \ = \ \dot{\lambda} x.\, \overline{\mathsf{fix}}_{\overline{\iota},\underline{\iota}}\cdot(\dot{\lambda} p.\, \dot{\lambda} n.\, \mathbf{if}\ n \overline{\equiv} 0\ \mathbf{then}\ \$\cdot 1\ \mathbf{else}\ x \,\underline{\times}\, p\cdot(n \,\overline{-}\, 1)) \\
\mathit{power}_{sd} : \overline{\iota} \dot{\rightarrow} \underline{\iota} \dot{\rightarrow} \underline{\iota} \ = \\
\qquad \dot{\lambda} x.\, \underline{\mathsf{fix}}_{\underline{\iota},\underline{\iota}}\cdot(\dot{\lambda} p.\, \dot{\lambda} n.\, \mathbf{if}\ n \,\underline{\equiv}\, \$\cdot 0\ \mathbf{then}\ \$\cdot 1\ \mathbf{else}\ \$\cdot x \,\underline{\times}\, p\cdot(n \,\underline{-}\, \$\cdot 1))
\end{array}$$

(Note that, unlike for the CBN variant, the "ifs" are not binding-time annotated.) Computing the normal form of $\bar{\lambda} x.\,\mathsf{power}_{ds} \cdot x \cdot 3$, we get

$$\bar{\lambda} g_0.\,\mathbf{let}\ g_1 = g_0 \mathbin{\underline{\times}} \$\cdot 1\ \mathbf{in\ let}\ g_2 = g_0 \mathbin{\underline{\times}} g_1\ \mathbf{in\ let}\ g_3 = g_0 \mathbin{\underline{\times}} g_2\ \mathbf{in}\ g_3$$

Conversely, if we specialize with respect to the base, by computing the normal form of $\bar{\lambda} n.\,\mathsf{power}_{sd} \cdot 5 \cdot n$, we obtain essentially just the let-normal form of the original program, with the literal argument 5 inlined:

$$\begin{array}{l}
\bar{\lambda} g_0.\,\mathbf{let}\ g_1 = \underline{\mathsf{fix}}_{\iota,\iota} \cdot (\lambda g_2.\,\lambda g_3.\,\mathbf{let}\ g_4 = (g_3 \mathbin{\underline{=}} \$\cdot 0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in\ if}\ x_4\ \mathbf{then}\ \$\cdot 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else\ let}\ g_5 = g_3 \mathbin{\underline{-}} \$\cdot 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in\ let}\ g_6 = g_2 \cdot g_5 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in\ let}\ g_7 = \$\cdot 5 \mathbin{\underline{\times}} g_6\ \mathbf{in}\ g_7) \\
\quad \mathbf{in\ let}\ g_8 = g_1 \cdot g_0\ \mathbf{in}\ g_8
\end{array}$$

Remark 9. Note that the normalized forms contain what seems to be excessive let-sequentialization of trivial arithmetic functions. This is because the residualizing interpretation specifically does not know anything about the intended evaluating interpretation of those constants, and in particular whether they might have computational effects that must not be reordered, discarded, or duplicated.

While the explicit naming of all intermediate results may be useful if the specialized programs are to be further machine-processed, it makes them hard to read for humans. Of course, one could unfold the "trivial" lets in a separate post-processing phase, but doing so loses some of the appeal of generating code directly from the semantics.

However, it is actually possible to annotate the types of "pure" arithmetic primitives to make them let-unfoldable at generation time, much as for CBN functions [Dan96]. Formally, such an annotation corresponds to imposing constraints on possible behaviors of the dynamic interpretations of $\times$, $=$, etc., so that these constants may only be interpreted as semantic functions that factor through the η of the dynamic monad.

5.5 Complete code for CBV normalization

For completeness, we include below the full code for the implementation of the CBV normalizer. The implementation of shift/reset in Figure 2 is equivalent to the one from [Fil99a], but streamlined a bit for SML/NJ's notion of first-class continuation. Figure 3 shows the implementation of type-indexed function families and presents the dynamic signature. Figure 4 shows the evaluating and residualizing interpretations of the dynamic signature, as well as an example of a binding-time separated term parameterized over the dynamic signature. (The function `power_ds` uses the evaluating interpretation explicitly, to highlight the parallels with `power_sd`. In practice, the static operations in `power_ds` would usually be expressed directly in terms of the corresponding native ML constructs.) Finally, Figure 5 shows a few concrete execution examples of evaluating and residualizing the power function.

```
functor Control (type ans) :
sig
   val reset : (unit -> ans) -> ans
   val shift : (('a -> ans) -> ans) -> 'a
end =
struct
   open SMLofNJ.Cont
   exception MissingReset
   val mk : ans cont option ref = ref NONE
   fun abort x =
      case !mk of SOME k => throw k x | NONE => raise MissingReset

   type ans = ans
   fun reset t =
      let val m = !mk
          val r = callcc (fn k => (mk := SOME k; abort (t ())))
      in mk := m; r end
   fun shift h =
      callcc (fn k => abort (h (fn v => reset (fn () => throw k v))))
end;

type var = string
datatype exp =
   VAR of var
 | CST of var
 | LIT_int of int
 | PAIR of exp * exp
 | TRUE
 | FALSE
 | LAM of var * exp
 | APP of exp * exp
 | LET of var * exp * exp
 | MATCH of (string * string) * exp * exp
 | IF of exp * exp * exp;

structure Aux :
sig
   val gensym : unit -> var
   val withct : (unit -> 'a) -> 'a
   val reset : (unit -> exp) -> exp
   val shift : (('a -> exp) -> exp) -> 'a
end =
struct
   val n = ref 0
   fun gensym () =
      let val x = "x" ^ (Int.toString (!n)) in n := (!n+1); x end
   fun withct t = let val on = !n
                  in n := 0; let val r = t () in n := on; r end end

   structure C = Control (type ans = exp)
   val reset = C.reset
   val shift = C.shift
end;
```

Fig. 2. Auxiliary definitions

```
structure Norm =
struct
   open Aux
   datatype 'a rr = RR of ('a -> exp) * (exp -> 'a)

   val dint = RR (fn e => e, fn e => e)
   val bool = RR (fn b => if b then TRUE else FALSE,
                  fn e => shift (fn k => IF (e, k true, k false)))
   fun prod (RR (rya, rta), RR (ryb, rtb)) =
        RR (fn p => PAIR (rya (#1 p), ryb (#2 p)),
            fn e => let val v1 = gensym ()
                        val v2 = gensym ()
                    in shift (fn k =>
                         MATCH ((v1,v2), e,
                                reset (fn () => k (rta (VAR v1),
                                                   rtb (VAR v2)))))
                    end)
   fun arrow (RR (rya, rta), RR (ryb, rtb)) =
        RR (fn f => let val v = gensym ()
                    in LAM (v, reset (fn () => ryb (f (rta (VAR v)))))
                    end,
            fn e => fn a => let val v = gensym ()
                            in shift (fn k =>
                                 LET (v, APP (e, rya a),
                                      reset (fn () => k (rtb (VAR v)))))
                            end)

   fun reify (RR (ry, rt)) a = withct (fn () => ry a)
   fun reflect (RR (ry, rt)) = rt
end;

signature DYN =
sig
   type dint
   type 'a rep

   val dint : dint rep
   val bool : bool rep
   val prod : 'a rep * 'b rep -> ('a * 'b) rep
   val arrow : 'a rep * 'b rep -> ('a -> 'b) rep

   val lift_int : int -> dint

   val plus : dint * dint -> dint
   val minus : dint * dint -> dint
   val times : dint * dint -> dint
   val equal : dint * dint -> bool
   val less : dint * dint -> bool

   val fix : ('a rep * 'b rep) -> (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
end;
```

Fig. 3. Normalization algorithm and dynamic signature

```
structure EvalD : DYN =
struct
   type dint = int
   type 'a rep = unit

   val dint = ()
   val bool = ()
   fun prod ((),()) = ()
   fun arrow ((),()) = ()

   fun lift_int n = n
   val plus = op +      val minus = op -      val times = op *
   val equal = op =     val less = op <

   fun fix ((),()) f = fn x => f (fix ((),()) f) x
end;

structure ResD : DYN =
struct
   open Aux Norm
   type dint = exp
   type 'a rep = 'a rr

   val lift_int = LIT_int
   val plus = reflect (arrow (prod (dint, dint), dint)) (CST "+")
   val minus = reflect (arrow (prod (dint, dint), dint)) (CST "-")
   val times = reflect (arrow (prod (dint, dint), dint)) (CST "*")
   val equal = reflect (arrow (prod (dint, dint), bool)) (CST "=")
   val less = reflect (arrow (prod (dint, dint), bool)) (CST "<")

   fun fix (rra, rrb) =
      reflect (arrow (arrow (arrow (rra, rrb), arrow (rra, rrb)),
                      arrow (rra, rrb)))
              (CST "fix")
end;

functor Power (D : DYN) =
struct
   fun power_ds x =
      EvalD.fix (EvalD.dint, EvalD.dint)
        (fn p => fn n =>
            if EvalD.equal (n, EvalD.lift_int 0)
            then D.lift_int 1
            else D.times (x, p (EvalD.minus(n, EvalD.lift_int 1))))

   fun power_sd x =
      D.fix (D.dint, D.dint)
        (fn p => fn n =>
            if D.equal (n, D.lift_int 0)
            then D.lift_int 1
            else D.times (D.lift_int x, p (D.minus (n, D.lift_int 1))))
end;
```

Fig. 4. Evaluating and residualizing realizations

```
structure PE = Power (EvalD);

val n1 = PE.power_ds (EvalD.lift_int 5) 3;
(* val n1 = 125 : EvalD.dint *)

val n2 = PE.power_sd 5 (EvalD.lift_int 3);
(* val n2 = 125 : EvalD.dint *)

structure PR = Power (ResD);

val t1 = Norm.reify (Norm.arrow (Norm.dint, Norm.dint))
                    (fn x => PR.power_ds x 3);
(*
val t1 =
  LAM
    ("x0",
     LET
       ("x1",APP (CST "*",PAIR (VAR "x0",LIT_int 1)),
        LET
          ("x2",APP (CST "*",PAIR (VAR "x0",VAR "x1")),
           LET ("x3",APP (CST "*",PAIR (VAR "x0",VAR "x2")),VAR "x3"))))
  : exp
*)

val t2 = Norm.reify (Norm.arrow (Norm.dint, Norm.dint))
                    (fn n => PR.power_sd 5 n);
(*
val t2 =
  LAM
    ("x0",
     LET
       ("x1",
        APP
          (CST "fix",
           LAM
             ("x2",
              LAM
                ("x3",
                 LET
                   ("x4",APP (CST "=",PAIR (VAR "x3",LIT_int 0)),
                    IF
                      (VAR "x4",LIT_int 1,
                       LET
                         ("x5",APP (CST "-",PAIR (VAR "x3",LIT_int 1)),
                          LET
                            ("x6",APP (VAR "x2",VAR "x5"),
                             LET
                               ("x7",
                                APP (CST "*",PAIR (LIT_int 5,VAR "x6")),
                                VAR "x7")))))))),
        LET ("x8",APP (VAR "x1",VAR "x0"),VAR "x8"))) : exp
*)
```

Fig. 5. Examples: evaluating and specializing power

5.6 Exercises

Exercise 7. Make the ML implementation of TDPE generate residual terms with explicit type tags for lambda-abstractions and polymorphic constants.

Exercise 8. Extend the ML implementation of TDPE with disjoint unions (sums).

Exercise 9. Extend TDPE to generate pattern-matching bindings for let and lambda instead of using an explicit match construct.

Exercise 10. The first Futamura projection is defined as the specialization of an interpreter with respect to a program [Fut71,Fut99]; it is a standard exercise in partial evaluation. In this open exercise, you are asked to write an interpreter for a simple imperative language, and to specialize it with respect to a program of your choice.

Specifically, you should write a core interpreter, in denotational style, as the body of a functor; this functor should be parameterized with the generic interpretation of each elementary operation that has to be happen at runtime (such as arithmetic, state lookup and modification, I/O operations, and fixed pointss for loops), much as in the power example. You should also write two structures: one for the static interpretations of all elementary constructs, and one for their dynamic interpretations. Again, as in the power example, instantiating the functor with the static structure should yield an interpreter, while instantiating it with the dynamic structure and should yield the core of a compiler.

Some inspiration for solving this exercise can be found in Danvy's lecture notes on type-directed partial evaluation [Dan98], in Grobauer and Yang's treatment of the second Futamura projection [GY01], and in Danvy and Vestergaard's take on semantics-based compiling by type-directed partial evaluation [DV96].

6 Summary and conclusions

We have shown two different versions of NBE in Sections 2 and 3, and in Sections 4 and 5 we showed how to generalize the idea of NBE to TDPE. Some of the key properties of NBE which we have exploited for TDPE are:

- Normal forms are characterized in terms of undirected equivalence, rather than directed reduction.
- The notion of equivalence is sound for equality with respect to a wide variety of interpretations.
- Among those interpretations, we pick a particular, quasi-syntactic one, which allows us to extract syntactic terms from denotations.

On the other hand, we also wish to emphasize some important adaptations and changes:

- TDPE introduces a notion of binding times, in the form of a distinction between interpreted (static) and uninterpreted (dynamic) base types and constants.

– We characterize equivalence in TDPE semantically (equality of interpretations in all models), rather than syntactically (convertibility).
– We consider a language with computational effects – either just potential divergence, or general monadic effects – not only a direct set-theoretic interpretation of functions.

Type-directed partial evaluation can be seen as a prime example of "applied semantics": while the basic TDPE algorithm, even for call-by-value, can be expressed in a few lines, we only get a proper understanding of how and why it works by considering its semantic counterpart.

It is somewhat surprising that the notion of normalization by evaluation is so robust and versatile. It may well be possible to find other instances within computer science – even outside of the field of programming-language theory – where an NBE view allows us to drastically simplify and speed up computations of canonical representatives of an equivalence class.

References

[ADHS01] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, Boston, Massachusetts, June 2001.

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference*, number 953 in Lecture Notes in Computer Science, Cambridge, UK, August 1995.

[AHS96] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalization for a polymorphic system. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 98–106, New Brunswick, New Jersey, July 1996.

[Asa02] Kenichi Asai. Binding-time analysis for both static and dynamic expressions. *New Generation Computing*, 20(1):27–51, 2002. Preliminary version available in the proceedings of SAS 1999 (LNCS 1694).

[Aug98] Lennart Augustsson. Cayenne – a language with dependent types. In *ACM SIGPLAN International Conference on Functional Programming* [ICF98], pages 239–250.

[Bar77] Henk Barendregt. The type free lambda calculus. In *Handbook of Mathematical Logic*, pages 1092–1132. North-Holland, 1977.

[Bar90] Henk Barendregt. Functional programming and lambda calculus. In *Handbook of Theoretical Computer Science*, pages 323–363. Elsevier, 1990.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[Ber93] 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.

[BS91] 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.

[CD93a] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[CD93b] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93*, number 806 in Lecture Notes in Computer Science, Nijmegen, The Netherlands, May 1993.

[CD97] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[ČDS98] Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

[Dan96] Olivier Danvy. Pragmatics of type-directed partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 73–94, Dagstuhl, Germany, February 1996. Springer-Verlag. Extended version available as the technical report BRICS RS-96-15.

[Dan98] 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.

[DD98] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Gothenburg, Sweden, May 8–9, 1998), number NS-98-8 in Note Series, Department of Computer Science, University of Aarhus, May 1998. BRICS.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990.

[DMP95] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.

[DV96] 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.

[Fil99a] 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.

[Fil99b] 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.

[Fil01] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Krakow, Poland, May 2001.

[Fut71] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):721–728, 1971. Reprinted in *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[Fut99] Yoshihiko Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, 1999.

[Gom91] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 12(4):147–172, April 1991.

[GP99] 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.

[GY01] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):173–219, 2001.

[ICF98] *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[JGS93] 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. Available electronically at `http://www.dina.dk/~sestoft/pebook/`.

[Laf88] Yves Lafont. *Logiques, Catégories et Machines*. PhD thesis, Université de Paris VII, Paris, France, January 1988.

[Mit96] John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.

[ML75a] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the 3rd Scandinavian Logic Symposium*, pages 81–109, 1975.

[ML75b] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[MN93] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93*, number 806 in Lecture Notes in Computer Science, pages 213–237, Nijmegen, The Netherlands, May 1993.

[Mog89] 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.

[Mog92] Torben Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, July 1992.

[NN88] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988.

[Oho99] Atsushi Ohori. A Curry-Howard isomorphism for compilation and program execution. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, number 1581 in Lecture Notes in Computer Science, pages 280–294, L'Aquila, Italy, April 1999.

[Pau00] Lawrence C. Paulson. Foundations of functional programming. Notes from a course given at the Computer Laboratory of Cambridge University, available from `http://www.cl.cam.ac.uk/users/lcp/papers/#Courses`, 2000.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, August 1972. Reprinted in *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[Ruf93] Erik Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, Stanford, California, February 1993. Technical report CSL-TR-93-563.

[SK01] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[SW74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2):135–152, April 2000.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, 1993.

[Yan98] Zhe Yang. Encoding types in ML-like languages. In *ACM SIGPLAN International Conference on Functional Programming* [ICF98], pages 289–300. Extended version to appear in TCS.