The Simple Type Theory of Normalisation by Evaluation

René Vestergaard*

CNRS-IML, Marseille, vester@iml.univ-mrs.fr, iml.univ-mrs.fr/~vester

Abstract. We develop the type theory of the Normalisation by Evaluation (NbE) algorithm for the λ-calculus in the simply-typed case. In particular, we show that the algorithm computes long $\beta(\eta)$-normal forms by means of Plotkin's call-by-name and call-by-value β-evaluation semantics. This is remarkable (i) as the algorithm decides full $\beta\eta$-equality and (ii) as the algorithm so-far only has been presented in highly technical model-theoretic terms. To showcase the effective means of the algorithm, we provide an environment machine implementation of the semantics. We also analyse the semantics and the environment machine in terms of strategies on the λ-calculus. The proof burden is slight.

1 Introduction

Normalisation by Evaluation (NbE) was first studied in its own right by Berger and Schwichtenberg about a decade ago [6]. It is a method by which the meaning function (aka evaluation functional) of a model of a language can be used to normalise terms of the language. Simply put, NbE is a type-indexed function which takes as argument a semantic object corresponding to some term and returns that term's long $\beta(\eta)$-normal form at the given type. All computation is performed in the model; still, the result is a syntactic term rather than a semantic object in the traditional sense. Because of this functionality, the model theory of NbE typically involves augmenting standard models with the relevant syntax; something which is highly non-trivial.

The NbE algorithm applies in many contexts and has, among other things, been presented as "an inverse to the evaluation functional" [6], "reduction-free normalisation" [1, 2], "type-directed partial evaluation" [10], as the computational content of Tait's typed method for proving β strongly normalising [5], as a glueing construction in intuitionistic model theory [7], and as pertaining to the Yoneda embedding in (constructive) category theory [9].

1.1 Implications

The original motivation for the NbE algorithm was largely practical [6]. The algorithm was invented to obtain syntactic representations of the internal proof

* Supported under EU TMR grant # ERBFMRXCT-980170: LINEAR.

objects of the MinLog proof system at LMU, Munich. The main focus in [6], apart from the domain-theoretic correctness proof, was thus the use of the quote and unquote constructs of the programming language Scheme [16] to provide a native (as opposed to interpretive) implementation of the algorithm for the simply-typed λ-calculus.[1] The non-trivial details of how such a native implementation works in a more traditional programming language setting are given in [12]. Related to this, [10, 11] substantiates the usefulness of NbE for purposes of partial evaluation and, in particular, semantics-based compiling (i.e., automated compiler derivation).

The main theoretical implication of NbE, apart from the number of its various technical incarnations, has so-far been considered to be the fact that the algorithm amounts to a non-rewriting-based decision procedure for typed $\beta\eta$-equivalence — the algorithm even computes the long $\beta(\eta)$-normal form of a term.

1.2 Our Contributions

We will show that NbE also has interesting theoretical implications in a rewriting-based setting. In particular, we will show that the algorithm computes long $\beta(\eta)$-normal forms by using very limited computational powers, i.e., Plotkin-style β-evaluation semantics [17], which makes it interesting from a practical, compiler-technology perspective as well. We accomplish this by introducing an evaluation-strategy indexed two-level λ-calculus: λ_b^{NbE} with $b \in \{\beta_{\mathrm{n}}, \beta_{\mathrm{v}}, \beta_{\mathrm{w}}\}$ for call-by-name (CBN), call-by-value (CBV), and their combination: call-by-whatever (CBW). We use $\lambda_{-}^{\mathrm{NbE}}$ to refer to all three in conjunction. One level of $\lambda_{-}^{\mathrm{NbE}}$ takes the role of the model, the other that of the target syntax. The syntactic level is evidently not subjected to evaluation. The core of the NbE algorithm is expressed as the canonical inter-level term coercers in $\lambda_{-}^{\mathrm{NbE}}$.

- We give a formal presentation of NbE which is straight to the point and which does not involve intricate model-theoretic technicalities.
- We present the type theory and Plotkin-style β-evaluation semantics of NbE and show (i) that the algorithm computes long $\beta(\eta)$-normal forms and (ii) that the semantics is sound and complete for simply-typed $\beta\eta$-equality.
- We present (and code in ML) a renaming-free environment machine which implements the type-theoretic semantics of NbE.
- We analyse the above in terms of reduction strategies in the λ-calculus.

Although the semantics-based approaches naturally use constructive models, we feel that our approach is superior in explaining the actual, effective means by which the computation of NbE is performed, e.g., via the environment machine. We hope this will help facilitate more practical programming language applications of NbE. Related to this, we can mention that we successfully have

[1] Native means that the computation needed to effectuate the algorithm can be and is undertaken by the evaluation mechanism of the implementing language.

$$\frac{}{x : \tau, \Gamma \triangleright x : \tau}\ (\mathrm{Var})$$

$$\frac{x : \tau, \Gamma \triangleright e : \sigma}{\Gamma \triangleright \lambda x.e : \tau \to \sigma}\ (\to \mathrm{I}) \qquad \frac{\Gamma \triangleright e_1 : \tau \to \sigma \qquad \Gamma \triangleright e_2 : \tau}{\Gamma \triangleright e_1 @ e_2 : \sigma}\ (\to \mathrm{E})$$

Fig. 1. Simple types for the λ-calculus — $\tau ::= o \mid \tau \to \tau$; for Γ, see Section 1.3.

$$\begin{aligned}
\lambda x.e &=_\alpha \lambda y.e[x := y] && \text{if } y \notin \mathrm{FV}(e)\\
(\lambda x.e_1) @ e_2 &=_\beta e_1[x := e_2] &&\\
e &=_\eta \lambda x.e @ x && \text{if } x \notin \mathrm{FV}(e) \ \&\ e \in \Lambda^{\tau\to\sigma}\\
=_{\alpha\beta\eta} &= \ =_\alpha \cup =_\beta \cup =_\eta &&
\end{aligned}$$

Fig. 2. The extensional equational theory of $\lambda^{\to}$ — $=_\alpha$, $=_\beta$, $=_\eta$, and $=_{\alpha\beta\eta}$ are taken to be congruences, $-[- := -]$ is non-capturing substitution, and $\mathrm{FV}(-)$ is *free variables*.

extended our set-up to System F, as a second to the unpublished [2]. We have also provided the first (interpreted) implementation (in ML) of NbE for System F via its environment machine, see the preliminary [19]. It is our impression that a native implementation of NbE for System F is not possible in any existing programming language. The reason is that a sub-term of a System F term can have its type changed by type instantiation (whereas the overall type naturally remains unchanged by subject reduction). This essentially means that type substitution needs to defined in a non-standard manner.

1.3 α-Equivalence and *All That*

We have attempted *not* to ignore the issue of variable names. In particular, we account explicitly for all α-equivalence in the statement of our theorems. One exception is that we have suppressed the variable-name details of the various typing relations as their precise form is secondary to our main focus. As it stands, terms like $\lambda x.\lambda x.x$ can, for example, not be typed. This could be overcome by including a weakening rule or a rule saying that α-equal terms have the same type. Similarly, we have not provided the exact format of the sequents of the typing judgements. Informally, we consider antecedents to be sets of variables (each related to a type) and impose well-formedness conditions. We refer the interested reader to [20] for an account of a formal development (of β-confluence) that attains the aforementioned level of precision.

1.4 β, η, and the λ-Calculus

A quick summary of the relevant features of the λ-calculus.

$$(\lambda x.e_1) @ e_2 \rightarrow_\beta e_1[x := e_2]$$

$$e \leftarrow_{\eta'} \lambda x.e @ x \quad \text{if } x \notin \mathrm{FV}(e) \ \&\ e \in \Lambda^{\tau\rightarrow\sigma} \ \&\ e \neq \lambda x'.e' \ \&\ e \text{ is not applied}$$

Fig. 3. β-reduction and restricted η-expansion *à la* Mints for $\lambda^\rightarrow$. The former is a congruence while the latter only is subjected to limited contextual closure as prescribed.

$$\begin{aligned} c &::= \lambda x.c \mid d \qquad (\text{if } d \in \Lambda^o, \text{ cf. Definition 1}) \\ d &::= x \mid d @ c \end{aligned}$$

Fig. 4. The pre-terms of $\lambda^\rightarrow$'s long $\beta(\eta)$-normal forms — d is for De-constructor (aka *neutral* terms) and c is for Constructor (aka *normal* terms).

Definition 1 *Let* $e ::= x \mid \lambda x.e \mid e @ e$ *be the pre-terms of the* simply-typed λ-calculus ($\lambda^\rightarrow$) *and, following Figure 1, let the terms be* $e \in \Lambda^\tau \Leftrightarrow^{\mathrm{def}} \exists \Gamma. \Gamma \triangleright e : \tau$. *The* extensional equational theory *of* $\lambda^\rightarrow$ *is* $\Lambda^- / =_{\alpha\beta\eta}$, *cf. Figure 2.*

The extensional equational theory of $\lambda^\rightarrow$ is axiomatised by several different rewriting relations, all with different properties. For example, unrestricted η-expansion (right-to-left orientation of the η-equation in Figure 2) is not strongly normalising. When adding a unit type, η-reduction (left-to-right orientation of the η-equation in Figure 2) destroys confluence when combined with β-reduction. The most robust axiomatisation seems to be β-reduction combined with what is known as *(Mints') restricted η-expansion*, cf. Figure 3.

Theorem 2 ($\lambda^\rightarrow$ is Extensionally Well-Behaved [4])

- **Rewriting** *:* $=_{\alpha\beta\eta}$ *is axiomatised by* $=_\alpha \cup \rightarrow_\beta \cup \leftarrow_{\eta'}$, *cf. Figure 3.*
- **Completeness** *:* $\rightarrow_\beta \cup \leftarrow_{\eta'}$ *is SN and confluent (on α-equivalence classes).*
- **Consistency** *:* $\Lambda^- / =_{\alpha\beta\eta}$ *is non-trivial.*
- **Decidability** *:* $=_{\alpha\beta\eta}$ *is decidable.*

It is known that restricted η-expansion is connected to Huet's long $\beta(\eta)$-normal forms [15] which enjoy an *analyticity property*: all typing is syntactically evident in the terms.

Definition 3 (Long $\beta(\eta)$-normal forms) *The* long $\beta(\eta)$-normal forms, $\Lambda^{\beta(\eta)-\mathrm{long}}$, *are the typable (cf. Figure 1) c's in Figure 4.*

Lemma 4 ([4]) $\Lambda^{\beta(\eta)-\mathrm{long}}$ *and the normal forms of* $\rightarrow_\beta \cup \leftarrow_{\eta'}$ *coincide.*

Furthermore, $\leftarrow_{\eta'}$ combined with $\rightarrow_\beta$ behaves well for other typing paradigms, e.g., unit types and System F^ω [4, 13]. Further considerations in categorical rewriting [4] and algebraic rewriting [8], have lead to restricted η-expansion assuming a *de facto* status as the choice η-relation for most typed purposes. That said, the study of the rewriting properties of the restricted η-relations is non-trivial.

1.5 CBN, CBV, and the λ-Calculus

In the seminal [17], Plotkin established that there are close ties between CBN and CBV β-evaluation of functional programming languages (in the form of the λ-calculus) and $\beta\eta$-equality in continuation-passing style (CPS) languages. The connection was later improved upon by Sabry and Felleisen in the context of control operators [18]. The gist of the work is that direct-style β-evaluation and CPS $\beta\eta$-equality are one and the same. On the one hand, this means that CBN and CBV can simulate each other. On the other hand, we know that the two set-ups are archetypical strands of compiler technology for functional languages and the results thus make it possible to formally reason about and compare the different ways of implementing functional languages [18]. These results have been further expanded up-on, e.g., by Hatcliff and Danvy [14] in the case of another common compiler technology for languages with control operators: thunks. The point we are making is that Plotkin's CBN and CBV β-evaluation semantics are integral parts of present-day compiler technology, even when not used directly.

1.6 Overview of This Paper

In Section 2, we introduce the type theory of NbE: the $\lambda_{\_}^{\mathrm{NbE}}$-calculi. In Section 3, we establish that the calculi indeed serve the intended NbE purpose. In Section 4, we present an environment machine implementation of the semantics of the calculi (i.e., of NbE). In Section 5, we present example runs of the algorithm/environment machine. Finally, in Section 6, we conclude.

1.7 Acknowledgements

We wish to thank Olivier Danvy and Laurent Regnier for fruitful discussions and Ulrich Berger, Martin Hofmann, and the anonymous referees for their comments. Thanks are also due to Paul Taylor whose TeX macros we are using.

2 The $\lambda_{\_}^{\mathrm{NbE}}$-Calculi

We now introduce our two-level $\lambda_{\_}^{\mathrm{NbE}}$-calculi with the intent of making them capture the type theory of the NbE algorithm under different evaluation paradigms. Some of the employed design techniques can be used to define a more general notion of two-level calculi. We have chosen to focus fairly narrowly on NbE.

The $\lambda_{\_}^{\mathrm{NbE}}$-calculi are constructed by taking a term model of $\lambda^{\rightarrow}$ (i.e., Plotkin-style β-evaluation semantics) and augmenting it with the relevant syntax, viz.

$$x \in \overline{\mathcal{VN}}_x,\ y \in \underline{\mathcal{VN}}_y,\ \overline{\mathcal{VN}}_x \cap \underline{\mathcal{VN}}_y = \emptyset,\ \mathcal{VN}^2 = \overline{\mathcal{VN}}_x \cup \underline{\mathcal{VN}}_y$$

$$\begin{array}{lll}
e ::= x \mid \overline{\lambda} x.e \mid e \,\overline{@}\, e & & \\
\quad\ \mid\ d & & (d \text{ subject to typing constraints, cf. Figure 6}) \\
c ::= \underline{\lambda} y.c \mid d & & (\text{if } d \in \underline{D}_o, \text{ cf. Definition 5}) \\
\quad\ \mid\ e & & (e \text{ subject to typing constraints, cf. Figure 6}) \\
d ::= y \mid d \,\underline{@}\, c & &
\end{array}$$

Fig. 5. The pre-terms of $\lambda_{\underline{-}}^{\mathrm{NbE}}$ over two-sorted variable names, $\mathcal{VN}^2$.

$$\frac{}{x \mathrel{\overline{:}} \tau, \Gamma; \Delta \rhd\!\rhd x \mathrel{\overline{:}} \tau}\ \overline{(\mathrm{Var})} \qquad\qquad \frac{}{\Gamma; \Delta, y \mathrel{\underline{:}} \tau \rhd\!\rhd y \mathrel{\underline{:}} \tau}\ \underline{(\mathrm{Var})}$$

$$\frac{x \mathrel{\overline{:}} \tau, \Gamma; \Delta \rhd\!\rhd e \mathrel{\overline{:}} \sigma}{\Gamma; \Delta \rhd\!\rhd \overline{\lambda} x.e \mathrel{\overline{:}} \tau \to \sigma}\ \overline{(\to \mathrm{I})} \qquad\qquad \frac{\Gamma; \Delta, y \mathrel{\underline{:}} \tau \rhd\!\rhd c \mathrel{\underline{:}} \sigma}{\Gamma; \Delta \rhd\!\rhd \underline{\lambda} y.c \mathrel{\underline{:}} \tau \to \sigma}\ \underline{(\to \mathrm{I})}$$

$$\frac{\Gamma; \Delta \rhd\!\rhd e_1 \mathrel{\overline{:}} \tau \to \sigma \qquad \Gamma; \Delta \rhd\!\rhd e_2 \mathrel{\overline{:}} \tau}{\Gamma; \Delta \rhd\!\rhd e_1 \,\overline{@}\, e_2 \mathrel{\overline{:}} \sigma}\ \overline{(\to \mathrm{E})} \qquad \frac{\Gamma; \Delta \rhd\!\rhd d \mathrel{\underline{:}} \tau \to \sigma \qquad \Gamma; \Delta \rhd\!\rhd c \mathrel{\underline{:}} \tau}{\Gamma; \Delta \rhd\!\rhd d \,\underline{@}\, c \mathrel{\underline{:}} \sigma}\ \underline{(\to \mathrm{E})}$$

$$\frac{\Gamma; \Delta \rhd\!\rhd d \mathrel{\underline{:}} o}{\Gamma; \Delta \rhd\!\rhd d \mathrel{\overline{:}} o}\ \overline{(\mathrm{U2O})} \qquad\qquad \frac{\Gamma; \Delta \rhd\!\rhd e \mathrel{\overline{:}} o}{\Gamma; \Delta \rhd\!\rhd e \mathrel{\underline{:}} o}\ \underline{(\mathrm{O2U})}$$

Fig. 6. $\lambda_{\underline{-}}^{\mathrm{NbE}}$'s two-tiered typing relation. Γ and Δ disjointly pertain to a typing level.

$\Lambda^{\beta(\eta)-\mathrm{long}}$. The former is written with overlines while the latter is written with underlines. Overlap between the two levels is only allowed at ground type. The reason is one of analyticity of the type-indexed NbE algorithm and is closely connected to the ground-type restriction on d's in long $\beta(\eta)$-normal forms, cf. Figures 4 and 5. This particular feature turns out to virtually be the defining property of NbE's type theory on which most of the other properties hinge.

Definition 5 (NbE Terms) *Following Figures 5 and 6, terms are as follows:*

$$\begin{aligned}
e \in \overline{\Lambda}_\tau^{\mathrm{NbE}} &\Leftrightarrow^{\mathrm{def}} \exists \Gamma, \Delta \,.\, \Gamma; \Delta \rhd\!\rhd e \mathrel{\overline{:}} \tau \\
c \in \underline{C}_\tau &\Leftrightarrow^{\mathrm{def}} \exists \Gamma, \Delta \,.\, \Gamma; \Delta \rhd\!\rhd c \mathrel{\underline{:}} \tau \\
d \in \underline{D}_\tau &\Leftrightarrow^{\mathrm{def}} \exists \Gamma, \Delta \,.\, \Gamma; \Delta \rhd\!\rhd d \mathrel{\underline{:}} \tau \\
\Lambda_\tau^{\mathrm{NbE}} &=^{\mathrm{def}} \overline{\Lambda}_\tau^{\mathrm{NbE}} \cup \underline{C}_\tau
\end{aligned}$$

We stress that we use two-sorted variable names in the NbE terms. The x's are used exclusively for the overlined level, the model, and the y's are used exclusively for the underlined level, the syntax.

$$\frac{}{(\overline{\lambda}x.e_1)\,\overline{@}\,e_2 \rightarrow_{\overline{\beta}_{\mathrm{n}}} e_1[x := e_2]} \qquad \frac{e_1 \rightarrow_{\overline{\beta}_{\mathrm{n}}} e_1'}{e_1\,\overline{@}\,e_2 \rightarrow_{\overline{\beta}_{\mathrm{n}}} e_1'\,\overline{@}\,e_2}$$

$$\frac{c \rightarrow_{\overline{\beta}_{\mathrm{n}}} c'}{\underline{\lambda}y.c \rightarrow_{\overline{\beta}_{\mathrm{n}}} \underline{\lambda}y.c'} \qquad \frac{d \rightarrow_{\overline{\beta}_{\mathrm{n}}} d'}{d\,\underline{@}\,c \rightarrow_{\overline{\beta}_{\mathrm{n}}} d'\,\underline{@}\,c} \qquad \frac{c \rightarrow_{\overline{\beta}_{\mathrm{n}}} c'}{d\,\underline{@}\,c \rightarrow_{\overline{\beta}_{\mathrm{n}}} d\,\underline{@}\,c'}$$

Fig. 7. CBN $\overline{\beta}$-evaluation.

$$\frac{v \in \{x, \overline{\lambda}x_0.e_0\} \cup \llcorner \Lambda^{\beta(\eta)-\mathrm{long}} \lrcorner}{(\overline{\lambda}x.e)\,\overline{@}\,v \rightarrow_{\overline{\beta}_{\mathrm{v}}} e[x := v]} \qquad \frac{e_1 \rightarrow_{\overline{\beta}_{\mathrm{v}}} e_1'}{e_1\,\overline{@}\,e_2 \rightarrow_{\overline{\beta}_{\mathrm{v}}} e_1'\,\overline{@}\,e_2} \qquad \frac{e_2 \rightarrow_{\overline{\beta}_{\mathrm{v}}} e_2'}{(\overline{\lambda}x.e_1)\,\overline{@}\,e_2 \rightarrow_{\overline{\beta}_{\mathrm{v}}} (\overline{\lambda}x.e_1)\,\overline{@}\,e_2'}$$

$$\frac{c \rightarrow_{\overline{\beta}_{\mathrm{v}}} c'}{\underline{\lambda}y.c \rightarrow_{\overline{\beta}_{\mathrm{v}}} \underline{\lambda}y.c'} \qquad \frac{d \rightarrow_{\overline{\beta}_{\mathrm{v}}} d'}{d\,\underline{@}\,c \rightarrow_{\overline{\beta}_{\mathrm{v}}} d'\,\underline{@}\,c} \qquad \frac{c \rightarrow_{\overline{\beta}_{\mathrm{v}}} c'}{d\,\underline{@}\,c \rightarrow_{\overline{\beta}_{\mathrm{v}}} d\,\underline{@}\,c'}$$

Fig. 8. CBV $\overline{\beta}$-evaluation.

Definition 6 *Let $\ulcorner - \urcorner$ be injection of ordinary λ-terms into the overlined level of the NbE terms and let $\llcorner - \lrcorner$ be injection into the underlined level.*

Before presenting the Plotkin-style evaluation semantics of the NbE terms, we point out that the notion of *value* we use for the CBV case, cf. Figure 8, does not match the standard notion exactly. In particular, we admit any strictly underlined term as a value (in the overlined level). This is done exclusively to simplify the proof of Lemma 16. In fact, it suffices to consider y's as the underlined values, as is standard. Still, the notion we use is obviously justified in the present set-up.

Definition 7 *Let λ_b^{NbE} be $(\Lambda_{-}^{\mathrm{NbE}}, \rightarrow_{\overline{b}})$ for $b \in \{\beta_{\mathrm{n}}, \beta_{\mathrm{v}}, \beta_{\mathrm{w}}\}$ with $\rightarrow_{\overline{\beta}_{\mathrm{n}}}$ and $\rightarrow_{\overline{\beta}_{\mathrm{v}}}$ given in Figures 7 respectively 8 and $\rightarrow_{\overline{\beta}_{\mathrm{w}}} = \rightarrow_{\overline{\beta}_{\mathrm{n}}} \cup \rightarrow_{\overline{\beta}_{\mathrm{v}}}$.*

Lemma 8 (Subject Reduction) *For $b \in \{\beta_{\mathrm{n}}, \beta_{\mathrm{v}}, \beta_{\mathrm{w}}\}$*

$$e \in \Lambda_\tau^{\mathrm{NbE}} \wedge e \rightarrow_{\overline{b}} e' \Rightarrow e' \in \Lambda_\tau^{\mathrm{NbE}}$$

Proof: Informally, it needs to be established (i) that the ground-type level-overlap property is not broken and (ii) that the syntactic constraints (long $\beta(\eta)$-normal formedness) of the underlined level are respected. The second, (ii), essentially follows from (i). In turn, (i) is established from the subject reduction

$$
\begin{array}{lll}
\downarrow^{\tau} \; : \; \overline{\Lambda}^{\mathrm{NbE}}_{\tau} \longrightarrow \underline{C}_{\tau} & & \text{(aka reify)} \\
\downarrow^{o} e = e & & \\
\downarrow^{\tau_1 \to \tau_2} e = \underline{\lambda} y. \downarrow^{\tau_2} (e \,\overline{@}\, (\uparrow_{\tau_1} y)) & & (\text{for } y \notin \mathrm{FV}(e)) \\
 & & \\
\uparrow_{\tau} \; : \; \underline{D}_{\tau} \longrightarrow \overline{\Lambda}^{\mathrm{NbE}}_{\tau} & & \text{(aka reflect)} \\
\uparrow_{o} d = d & & \\
\uparrow_{\tau_1 \to \tau_2} d = \overline{\lambda} x. \uparrow_{\tau_2} (d \,\underline{@}\, (\downarrow^{\tau_1} x)) & & (\text{for } x \notin \mathrm{FV}(d))
\end{array}
$$

Fig. 9. Level coercers for the $\lambda^{\mathrm{NbE}}_{-}$-calculi.

property of the simply-typed λ-calculus by observing that types of (residuals of) sub-terms and not merely of terms proper are preserved under reduction.[2] □

As an interlude, we can show that $\lambda^{\mathrm{NbE}}_{-}$ is sound with respect to $\lambda^{\to}$.

Definition 9 *Let $|-| : \Lambda^{\mathrm{NbE}}_{\tau} \longrightarrow \Lambda^{\tau}$ be the function that strips off over- and underlines.*

Lemma 10 *For $b \in \{\beta_{\mathrm{n}}, \beta_{\mathrm{v}}, \beta_{\mathrm{w}}\}$, $\to_{\overline{b}}$ respects $\to_{\beta}$ under $|-|$:*

$$e \to_{\overline{b}} e' \Rightarrow |\, e \,| \to_{\beta} |\, e' \,|$$

Proof: Straightforward. □

Two easy consequences of Lemma 10 following Theorem 2 are:

Lemma 11 (SN) *For $b \in \{\beta_{\mathrm{n}}, \beta_{\mathrm{v}}, \beta_{\mathrm{w}}\}$, $\to_{\overline{b}}$ is SN.*

Lemma 12 (Consistency) *The equational theories of $\lambda^{\mathrm{NbE}}_{-}$ are non-trivial.*

2.1 Type-Indexed Inter-Level Coercion in $\lambda^{\mathrm{NbE}}_{-}$

We will now derive the core of the NbE algorithm, *reflect* and *reify*, as the canonical inter-level term coercers in $\lambda^{\mathrm{NbE}}_{-}$, cf. Figure 9. Informally, coercion is:

- source-level application of function-typed terms till reaching ground type,
- actual term-transferral at ground type using $\overline{\text{(U2O)}}$ and $\underline{\text{(O2U)}}$, and
- target-level λ-abstraction corresponding to function types.

In order to introduce coercion, we consider two examples. In Section 5, we will revisit these examples with the help of the environment machine we present.

[2] This is where the simple and polymorphic type theory of NbE part ways.

Example: Simple Function Type Let us consider $\overline{\lambda}x.x$ at type $o \to o$. By applying the term to, say, y at type o (an underlined variable of ground type may occur in both levels), we get a term of ground type which also is allowed to occur in both levels; in particular, it may occur as the body of an underlined abstraction over y: $\underline{\lambda}y.(\overline{\lambda}x.x) \mathbin{\overline{@}} y$. Type theoretically this reads:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x \mathrel{\overline{:}} o ; y \mathrel{\underline{:}} o \rhd\!\!\rhd x \mathrel{\overline{:}} o}\,\overline{(\mathrm{Var})}}{; y \mathrel{\underline{:}} o \rhd\!\!\rhd \overline{\lambda}x.x \mathrel{\overline{:}} o \to o}\,\overline{(\to \mathrm{I})} \qquad \dfrac{\dfrac{}{; y \mathrel{\underline{:}} o \rhd\!\!\rhd y \mathrel{\underline{:}} o}\,\underline{(\mathrm{Var})}}{; y \mathrel{\underline{:}} o \rhd\!\!\rhd y \mathrel{\overline{:}} o}\,\overline{(\mathrm{U2O})}}{; y \mathrel{\underline{:}} o \rhd\!\!\rhd (\overline{\lambda}x.x) \mathbin{\overline{@}} y \mathrel{\overline{:}} o}\,\overline{(\to \mathrm{E})}}{; y \mathrel{\underline{:}} o \rhd\!\!\rhd (\overline{\lambda}x.x) \mathbin{\overline{@}} y \mathrel{\underline{:}} o}\,\underline{(\mathrm{O2U})}}{; \ \rhd\!\!\rhd \underline{\lambda}y.(\overline{\lambda}x.x) \mathbin{\overline{@}} y \mathrel{\underline{:}} o \to o}\,\underline{(\to \mathrm{I})}$$

The constructed term normalises to $\underline{\lambda}y.y$ under all evaluation paradigms, cf. Definition 7.

Example: Negative Function Type Consider $\overline{\lambda}x.x$ at type $(o \to o) \to o \to o$. We can bring the term into the underlined level by the above trick applied twice if we, starting from an underlined variable (y_1 below), can construct a term at (the negatively occurring) type $o \to o$ which belongs to the overlined level. We dualise the employed trick and get:

$$\underline{\lambda}y_1.\underline{\lambda}y_2.((\overline{\lambda}x.x) \mathbin{\overline{@}} (\overline{\lambda}x_1.y_1 \mathbin{\underline{@}} x_1)) \mathbin{\overline{@}} y_2$$

The normal form of the constructed term is $\underline{\lambda}y_1.\underline{\lambda}y_2.y_1 \mathbin{\underline{@}} y_2$ under all evaluation paradigms.

The two normal forms we have presented are easily seen to be long $\beta(\eta)$-normal forms of the originating terms at the given type — and to exist in the other level.

Lemma 13 *$\downarrow^{\tau} \colon \overline{\Lambda}_{\tau}^{\mathrm{NbE}} \longrightarrow \underline{C}_{\tau}$ (reify) and $\uparrow_{\tau} \colon \underline{D}_{\tau} \longrightarrow \overline{\Lambda}_{\tau}^{\mathrm{NbE}}$ (reflect), cf. Figure 9, are well-defined (given a way of picking new names).*

Proof: By a simple induction on typing derivations. □

On the issue of a correct way of picking fresh variable names for reify and reflect, we point out that there is a simple solution in our set-up with two-sorted variable names. As we shall see, $\downarrow^{-}$ and $\uparrow_{-}$ are only applied in a very specific manner, cf. Theorem 17. In particular, their application always starts with $\downarrow^{-}$ being applied to a completely overlined term which thus contains no y's. The y's can therefore simply be picked in order starting from "the first" — that is, $\underline{\mathcal{VN}_y}$ is taken to be countably infinite. As for the x's, we notice that while reify (i.e., $\downarrow^{-}$) is applied to arbitrary overlined terms, reflect (i.e., $\uparrow_{-}$) is only applied to underlined terms that are themselves constructed by reify and reflect, cf. the definition of $\downarrow^{\tau_1 \to \tau_2}$, Figure 9. Differently said, any abstraction over an x we have picked will only have other fresh x's (and y's) in its scope. Hence, they too can be picked starting from "the first" — that is, $\overline{\mathcal{VN}}_x$ is taken to be countably infinite as well. No inspection of terms is needed as long as we maintain a counter of the last picked x and/or y. Had we instead considered one-sorted variable names, it would have been necessary to ascertain which is the highest index'ed free variable in the term we initially apply reify to and start picking the y's above that. That said, the use of two-sorted variable names is obviously justified for NbE.

Lemma 14 *$\downarrow^\tau$ and $\uparrow_\tau$ respect $=_\eta$ under $|-|$:*

$$\begin{array}{r} \forall e \in \overline{\Lambda}_\tau^{\mathrm{NbE}} . \mid e \mid =_\eta \mid \downarrow^\tau e \mid \\ \forall d \in \underline{D}_\tau . \mid d \mid =_\eta \mid \uparrow_\tau d \mid \end{array}$$

Proof: By a simple induction on types. □

We see that the effect of $\downarrow^\tau$ and $\uparrow_\tau$ does not respect $\leftarrow_{\eta'}$ under $|-|$ as the functions can be applied to abstractions. On the other hand, Mints' restrictions serve to make η-expansion terminating while $\downarrow^\tau$ and $\uparrow_\tau$ are well-defined in themselves by virtue of being defined inductively on types.

3 Normalisation by Evaluation

With the purpose-defined $\lambda_{-}^{\mathrm{NbE}}$ in place, the task of presenting the NbE algorithm is almost trivial. For clarity of presentation, we will initially restrict attention to closed terms but will go on to address NbE for arbitrary open terms immediately afterwards.

Definition 15 (x-Closed Terms)

- *Let $\widehat{\Lambda^\tau}$ be the subset of Λ^τ of terms with no free variables.*
- *Let $\overset{\widehat{x}}{\overline{\Lambda}}{}_\tau^{\mathrm{NbE}}$ be the subset of $\overline{\Lambda}_\tau^{\mathrm{NbE}}$ of terms with no free x's.*
- *Let $\overset{\widehat{x}}{\underline{C}}_\tau$ be the subset of $\underline{C}_\tau$ of terms with no free x's.*

The following result, the only non-standard result we need, essentially says that reduction in $\lambda_{-}^{\mathrm{NbE}}$ is capable of expressing the kind of semantics-to-syntax computation we need in order to perform NbE.

Lemma 16 (Overline-Free Normal Forms) *Let $b \in \{\beta_{\mathrm{n}}, \beta_{\mathrm{v}}, \beta_{\mathrm{w}}\}$*

$$\forall c \in \overset{\widehat{x}}{\underline{C}}_\tau \,.\, c \textit{ is } \rightarrow_{\overline{b}}\textit{-normal} \;\Rightarrow\; c \in \llcorner \Lambda^{\beta(\eta)-\mathrm{long}} \lrcorner$$

Proof: The result is proved by a double structural induction (on terms) alongside the following property:[3]

$$\forall e \in \overset{\widehat{x}}{\overline{\Lambda}}{}_\tau^{\mathrm{NbE}} \,.\, e \text{ is } \rightarrow_{\overline{b}}\text{-normal} \;\Rightarrow\; (e \in \overset{\widehat{x}}{\underline{C}}_o \,\vee\, e \equiv \overline{\lambda} x.e')$$

[3] In fact, it is a triple structural induction, with a case for d as well — we thus use *the* primitive induction principle for the pre-terms of $\Lambda_\tau^{\mathrm{NbE}}$, cf. Figure 5.

The only non-trivial case is $e \equiv e_1 \,\overline{@}\, e_2$ for the latter property. Assume e is $\rightarrow_{\overline{b}}$-normal. By definition of $\rightarrow_{\overline{b}}$, e_1 is $\rightarrow_{\overline{b}}$-normal and by I.H., $e_1 \in \underline{\mathrm{C}}_o^{\widehat{x}} \vee e_1 \equiv \overline{\lambda} x.e_1'$. The first disjunct is not possible by the typing constraints on applications. In the case of the second disjunct, e is a redex and thus not $\rightarrow_{\overline{\beta}_\mathrm{n}}$-normal which means that we are done by contradiction. Consider, now, the situation for $\rightarrow_{\overline{\beta}_\mathrm{v}}$. By definition, e_2 must be normal and thus $e_2 \in \underline{\mathrm{C}}_o^{\widehat{x}} \vee e_2 \equiv \overline{\lambda} x.e_2'$ by I.H. For the first disjunct, we have that $e_2 \in \llcorner \Lambda^{\beta(\eta)-\mathrm{long}} \lrcorner$ by the other I.H. and e is a $\rightarrow_{\overline{\beta}_\mathrm{v}}$-redex, contradicting normality. Similarly for the other disjunct. The case for $\rightarrow_{\overline{\beta}_\mathrm{w}}$ follows. □

We are now ready to normalise $\lambda^{\rightarrow}$ by evaluation. Before doing so we stress that $\rightarrow_{\overline{\beta}_\mathrm{n}}$ and $\rightarrow_{\overline{\beta}_\mathrm{v}}$ admit at most one redex in an overlined term by definition (and thus at most as many redexes as there are outermost overlined terms in an underlined term). If present, the redex will be located in a pre-determined position. In fact, by Lemma 16 there will always be a redex in that position or the term will be underlined. No inspection of overlined terms is thus needed to perform a $\rightarrow_{\overline{b}}$ step and, furthermore, reduction will always terminate. This comment will be further explored in Sections 3.1 and 4.1.

Theorem 17 (Normalisation by Evaluation) *Let $b \in \{\beta_\mathrm{n}, \beta_\mathrm{v}, \beta_\mathrm{w}\}$*

$$\forall e \in \widehat{\Lambda}^\tau . \exists c \in \Lambda^{\beta(\eta)-\mathrm{long}} . e =_{\beta\eta} c \;\wedge\; ((\downarrow^\tau \ulcorner e \urcorner) \twoheadrightarrow_{\overline{b}} \llcorner c \lrcorner)$$

Proof: From Lemmas 10, 11, 14, and 16. □

Proposition 18 *NbE, as presented in Theorem 17 (but up-to α), can be extended to open terms (i.e., to Λ^τ) if the free x's and their type are provided.*

Proof: Abstract the free variables with λ's before applying $\downarrow^{-} \ulcorner - \urcorner$ at the relevant type (and rename the extra abstractions back to what they were and then strip them off of c). □

This is not the most elegant way of extending NbE to open terms and we will present a more pleasing way of performing the extension later. Still, we can make the following observation on our way to show that NbE decides $\beta\eta$-equality.

Proposition 19 $\mathrm{NbE}^\tau : \Lambda^\tau \longrightarrow \Lambda^{\beta(\eta)-\mathrm{long}}$ *($e \mapsto c$, cf. Theorem 17) is functional up-to α and given the type of free variables.*

Theorem 20 (NbE is Sound and Complete for $=_{\beta\eta}$ in $\lambda^{\rightarrow}$)

$$\forall e_1, e_2 \in \Lambda^\tau \,.\, e_1 =_{\alpha\beta\eta} e_2 \Leftrightarrow \mathrm{NbE}^\tau(e_1) =_\alpha \mathrm{NbE}^\tau(e_2)$$

The reason the theorem is "up-to α" is that long $\beta(\eta)$-normal forms only are unique up-to α, cf. Theorem 2, Completeness.

3.1 The Reduction Strategy of $|\,\underline{\lambda}^{\mathbf{NbE}}\,|$ — I

The following definition is inspired by [3, Chapter 13: "Reduction Strategies"].[4] Some notions will not be used until Section 4.1.

Definition 21 (Strategies) *A map, $\mathcal{S} : A \longrightarrow A$, is a 1-strategy with respect to $\rightarrow_A \subseteq A \times A$ if $a \rightarrow_A \mathcal{S}(a)$. It is a strategy if $a \twoheadrightarrow_A \mathcal{S}(a)$. A (1-)strategy is effective if it is not only recursive but defined by induction on terms. It is normalising if $e \twoheadrightarrow_A e' \wedge (e' \text{ is } \rightarrow_A -\text{normal}) \Rightarrow \exists n.\mathcal{S}^n(e) = e'$. It is 1-normalising if $n = 1$.*

With the definition in place, we can immediately present the following result which clarifies our remarks prior to Theorem 17 as well as justifying and introducing the next section. It relies on the underlined level being immutable as far as evaluation goes. This means that evaluation in the two subterms of an underlined application is strictly parallel and cannot interfere with each other.

Lemma 22 *Adjust the relevant rule of $\rightarrow_{\overline{\beta}_{\mathrm{n}}}$ to:*

$$\frac{d \text{ is } \rightarrow_{\overline{\beta}_{\mathrm{n}}} -\text{normal} \qquad c \rightarrow_{\overline{\beta}_{\mathrm{n}}} c'}{d \mathbin{\underline{@}} c \rightarrow_{\overline{\beta}_{\mathrm{n}}} d \mathbin{\underline{@}} c'}$$

Up-to $|-|$, $\rightarrow_{\overline{\beta}_{\mathrm{n}}}$ (adjusted) is an effective, normalising 1-strategy for $\rightarrow_\beta$ in $\lambda^{\rightarrow}$.

Proof: The relation is the leftmost β-reduction strategy of $\lambda^{\rightarrow}$ up-to $|-|$ and the result follows from [3, Theorem 13.2.2]. $\square$

4 A Renaming-Free Environment Machine

We will present an environment machine that implements $\lambda^{\mathrm{NbE}}_{\beta_{\mathrm{n}}}$. A similar machine can be defined for $\lambda^{\mathrm{NbE}}_{\beta_{\mathrm{v}}}$. The fact that the machine is an environment machine, and thus never performs (full-fledged) substitution but only *instantiates* variables not under an abstraction, follows from the *evaluation* semantics of NbE. A subtle point is thus that the machine never performs any renaming of bound variables to maintain correct binding under reduction. Instead, the structure of the environments subsumes the scoping rules of $\lambda^{\mathrm{NbE}}_{\beta_{\mathrm{n}}}$.

Note 23 (Term Environments)

$\perp$ *is the no-where defined term environment, cf. Figure 10.*
$-[- \mapsto -]$ *is environment augmentation; it shadows existing bindings.*

[4] We use "inductive on terms" rather than Barendregt's "[computable] in a relatively simple way" to underpin "effective". We also introduce "1-normalising".

$$\rho \in \text{TermEnv} ::= \overline{\mathcal{VN}}_x \rightharpoonup \overline{\Lambda}^{\text{NbE}}_{\_} \times \text{TermEnv}$$

$$\frac{\rho(x) = \langle e', \rho' \rangle \qquad \langle e', \rho' \rangle \Downarrow^{\text{o}} \langle e'', \rho'' \rangle}{\langle x, \rho \rangle \Downarrow^{\text{o}} \langle e'', \rho'' \rangle} \qquad \frac{}{\langle \overline{\lambda} x.e, \rho \rangle \Downarrow^{\text{o}} \langle \overline{\lambda} x.e, \rho \rangle}$$

$$\frac{\langle e_1, \rho \rangle \Downarrow^{\text{o}} \langle \overline{\lambda} x.e', \rho' \rangle \qquad \langle e', \rho'[x \mapsto \langle e_2, \rho \rangle] \rangle \Downarrow^{\text{o}} \langle e'', \rho'' \rangle}{\langle e_1 \mathbin{\overline{@}} e_2, \rho \rangle \Downarrow^{\text{o}} \langle e'', \rho'' \rangle} \qquad \frac{\langle d, \rho \rangle \Downarrow^{\text{u}} d'}{\langle d, \rho \rangle \Downarrow^{\text{o}} \langle d', \bot \rangle}$$

$$\frac{}{\langle y, \rho \rangle \Downarrow^{\text{u}} y} \qquad \frac{\langle c, \rho \rangle \Downarrow^{\text{u}} c'}{\langle \underline{\lambda} y.c, \rho \rangle \Downarrow^{\text{u}} \underline{\lambda} y.c'} \qquad \frac{\langle d, \rho \rangle \Downarrow^{\text{u}} d' \qquad \langle c, \rho \rangle \Downarrow^{\text{u}} c'}{\langle d \mathbin{\underline{@}} c, \rho \rangle \Downarrow^{\text{u}} d' \mathbin{\underline{@}} c'} \qquad \frac{\langle e, \rho \rangle \Downarrow^{\text{o}} \langle e', \rho' \rangle}{\langle e, \rho \rangle \Downarrow^{\text{u}} e'}$$

Fig. 10. The NbE machine — an environment machine implementing $\lambda^{\text{NbE}}_{\beta_{\text{n}}}$.

Theorem 24 (The NbE Machine) $\langle -, \bot \rangle \Downarrow^{\text{u}} - : \underline{C}_\tau \rightharpoonup \underline{C}_\tau$ *as defined in Figure 10 is a (partial) function which preserves* $=_{\beta\eta}$ *under* $|-|$*. It is total on* $\overset{\widehat{x}}{\underline{\text{C}}}_\tau$*. In that case, it computes long* $\beta(\eta)$*-normal forms.*

Proof: Functionality follows as $\Downarrow^{\text{u}}$ and $\Downarrow^{\text{o}}$ are given inductively. Totality follows (i) from x-closedness making ρ suitably defined in the top-left rule and (ii) from a simple adaptation of the proof of Lemma 16: the left premise on the β_{n}-contraction rule is always met. The last point follows from Theorem 17. □

Proposition 25 (Open Terms) *The NbE Machine is total on* $\underline{C}_\tau$ *if, for a given term, we take* $\bot$ *to send any free variable,* x*, to* $\uparrow_\tau y$ *for a fresh* y *and the right* τ*. The resulting term with these* y*'s changed back to the* x*'s is a long* $\beta\eta$*-normal form of the original term.*

We have implemented (straightforwardly) the environment machine in Figure 10 in Moscow ML, cf. Appendix A. We refer the reader to our homepage for a downloadable version. The next section will present example runs of it. First, however, we finish our discussion of the reduction strategy underlying NbE.

4.1 The Reduction Strategy of $|\, \lambda^{\text{NbE}}_{\_} \,|$ — II

Following on from Section 3.1, we have the following result.

Lemma 26 *At each* τ, NbE^τ *is an effective, 1-normalising strategy for* $\rightarrow_\beta$*.*

Proof: The fact that NbE^τ is effective follows from Proposition 25. The other properties follow directly from Lemma 22. $\square$

The main point of interest in the lemma is the fact that NbE is effective. It means that NbE does not require extensive and repeated search for redexes but rather is defined as a one-step relation directly by induction on terms. It also means that the overlined level is never inspected, only tested for sort, so to speak. In particular, it is never necessary to "look inside" a closure, i.e., an overlined abstraction paired with an environment.

5 Examples

Our motivating examples for the definition of reify and reflect, cf. Section 2.1, were the reification of the identity function at different types. We now re-do these and other examples using our implementation of the NbE machine, cf. Appendix A. All code is available from our homepage. The output has been edited for space, only. `pp` is for pretty print, double constructors (e.g., @@ for application) belong to the overlined level, and single constructors belong to the underlined level.

First, we reify the identity function at increasingly more complex types:

1. ```
   - pp_o idterm;
   > val it = "(lla x1 . x1)" : string
   - pp_type idtype1;
   > val it = "(a1 -> a1)" : string
   - nbe idtype1 idterm;
   > val it = "(la y1 . y1)" : string
   ```
2. ```
   - pp_type idtype2;
   > val it = "((a1 -> a1) -> (a1 -> a1))" : string
   - nbe idtype2 idterm;
   > val it = "(la y1 . (la y3 . (y1 @ y3)))" : string
   ```
3. ```
   - pp_type idtype3;
   > val it = "((a1 -> (a1 -> a1))
                -> (a1 -> (a1 -> a1)))" : string
   - nbe idtype3 idterm;
   > val it = "(la y1 . (la y4 .
                 (la y5 . ((y1 @ y4) @ y5))))" : string
   ```
4. ```
   - pp_type idtype4;
   > val it = "(((a1 -> a1) -> (a1 -> a1))
                 -> ((a1 -> a1) -> (a1 -> a1)))" : string
   - nbe idtype4 idterm;
   > val it = "(la y1 . (la y5 . (la y7 .
                 ((y1 @ (la y3 . (y5 @ y3))) @ y7))))" : string
   ```

Second, we do arithmetic with Church Numerals, all at the same type:

```
- pp_type CNtype;
> val it = "((a1 -> a1) -> (a1 -> a1))" : string
- pp_o add;
> val it = "(lla x1 . (lla x2 . (lla x3 . (lla x4 .
            ((x1 @@ x3) @@ ((x2 @@ x3) @@ x4))))))" : string
- pp_o mult;
> val it = "(lla x1 . (lla x2 . (lla x3 .
            (x1 @@ (x2 @@ x3)))))" : string
- pp_o exp;
> val it = "(lla x1 . (lla x2 . (x1 @@ x2)))" : string
```

With the various definitions done as suggested, we thus have:

zero :

```
- pp_o zero;
> val it = "(lla x1 . (lla x2 . x2))" : string
- nbe CNtype zero;
> val it = "(la y1 . (la y3 . y3))" : string
```

one :

```
- pp_o one;
> val it = "(lla x1 . (lla x2 . (x1 @@ x2)))" : string
- nbe CNtype one;
> val it = "(la y1 . (la y3 . (y1 @ y3)))" : string
```

two = (add one) one :

```
- nbe CNtype two;
> val it = "(la y1 . (la y3 . (y1 @ (y1 @ y3))))" : string
```

three = (add one) two :

```
- nbe CNtype three;
> val it = "(la y1 . (la y3 .
             (y1 @ (y1 @ (y1 @ y3)))))" : string
```

four = (mult two) two :

```
- nbe CNtype four;
> val it = "(la y1 . (la y3 .
             (y1 @ (y1 @ (y1 @ (y1 @ y3))))))" : string
```

eight = (exp three) two :

```
- nbe CNtype eight;
> val it =
    "(la y1 . (la y3 .
       (y1@(y1@(y1@(y1@(y1@(y1@(y1@(y1@y3))))))))))" : string
```

6 Conclusion

We have, as a first, presented the type theory of the well-studied NbE algorithm by means of a Plotkin-style β-evaluation semantics. The semantics was implemented with an environment machine. Succinctly stated, our main result is that the algorithm *effectively* computes long $\beta(\eta)$-normal forms, and thus decides full $\beta\eta$-equality in the simply-typed λ-calculus, by means of β-evaluation only. No η-reduction is employed. Instead, we defined a "term skeleton", $\downarrow^{-}$, at each type which is wrapped around the considered term. It serves to bring all redexes into head position relative to the underlined level.

Future work includes the treatment of System F [19] and the determination of the extent of soundness and completeness of the algorithm when applied to terms that are not typable. This latter line of work follows from the observation that termination of the algorithm only requires the existence of a well-typed long $\beta(\eta)$-normal form and not well-typedness of the considered term itself. Formally, this follows from the fact that the implemented reduction strategy is normalising, cf. Sections 3.1 and 4.1.

References

1. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.
2. Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for system F. Manuscript, 1996.
3. Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics (Revised Edition)*. North-Holland, 1984.
4. C. Barry Jay and Neil Ghani. The virtues of eta-expansion. *Journal of Functional Programming*, 5(2):135–154, 1995.
5. 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.
6. 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. IEEE Computer Society Press.
7. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:73–94, 1997.
8. Roberto Di Cosmo and Delia Kesner. Combining algebraic rewriting, extensional lambda calculi, and fixpoints. *Theoretical Computer Science*, 169(2):201–220, 5 December 1996.
9. Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8(2):153–192, April 1998.

10. Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.
11. Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Extended version available as the technical report BRICS-RS-96-13.
12. Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan 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. Springer-Verlag. Extended version available as the technical report BRICS RS-99-17.
13. Neil Ghani. Eta-expansions in F^ω. In Dirk van Dalen and Marc Bezem, editors, *CSL-10*, LNCS 1258, pages 182–197, 1996.
14. John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, May 1997.
15. Gérard Huet. Résolution d'équations dans les langages d'ordre 1, 2, ..., ω. Thèse d'État, Université de Paris VII, Paris, France, 1976.
16. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.
17. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
18. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, December 1993.
19. René Vestergaard. The polymorphic type theory of Normalisation by Evaluation (preliminary version). Available at the author's homepage, 2001.
20. René Vestergaard and James Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names (*Barendregt was right after all ... almost*). In *RTA-12*, 2001. LNCS.

A The NbE Machine in (Moscow) ML

A.1 Abstract Syntax

```
infixr -->;

type T_var = int;

datatype Ty = tvar of T_var
            | --> of Ty * Ty;

type O_var = int;
```

```
type U_var = int;

datatype O_term = ovar of O_var
                | oabstr of O_var * O_term
                | oappl of O_term * O_term
                | u2o of U_deconst
and U_const     = uabstr of U_var * U_const
                | deconst of U_deconst
                | o2u of O_term
and U_deconst   = uvar of U_var
                | uappl of U_deconst * U_const;

val voidtype = tvar ~1;
val voidterm = ovar ~1;
```

A.2 Up and Down Arrows and Evaluation

```
load "Int";

val counter = ref 0;

fun newindex () = let val _ = counter := !counter + 1
                   in !counter
                  end;

fun setcounter n = counter := n;

use "abstract-syntax.sml";
use "pretty-print.sml";

datatype Envval = envval of O_term * (int -> Envval);

exception unbound_variable of string;

exception ill_formed_term_or_wrong_type of string;

val init_env =
    (fn var => raise unbound_variable ("** "^(Int.toString var)));

fun aug_env tenv var closure =
  (fn var2 => if var=var2
   then closure
   else (tenv var2) handle unbound_variable str
                   => raise unbound_variable ((Int.toString var) ^ " " ^ str));

fun down (tvar var) e =
```

```
      o2u e
  | down (t1 --> t2) e =
      let val var = newindex()
       in uabstr (var,(down t2 (oappl (e, up t1 (uvar var)))))
      end
and up (tvar var) d =
      u2o d
  | up (t1 --> t2) d =
      let val var = newindex()
       in oabstr (var,(up t2 (uappl (d, down t1 (ovar var)))))
      end;

fun o_eval (ovar ov) term_env =
      let val envval (ot,term_env1) = (term_env ov)
       in o_eval ot term_env1
      end
  | o_eval (oabstr (ov,ot)) term_env =
      ((oabstr (ov,ot)),term_env)
  | o_eval (oappl (ot1,ot2)) term_env =
      (let val (ot3,term_env1) = o_eval ot1 term_env
        in (case ot3
             of (oabstr (ov,ot4))
                  => o_eval ot4
                           (aug_env term_env1 ov (envval (ot2,term_env)))
              | _ => raise ill_formed_term_or_wrong_type ((pp_o ot3)
                            ^ " ** excepted abstraction at fct type"))
       end)
  | o_eval (u2o ud) term_env =
      (u2o (ud_eval ud term_env),init_env)
and uc_eval (uabstr (uv,uc)) term_env =
       uabstr (uv,(uc_eval uc term_env))
  | uc_eval (deconst ud) term_env =
       deconst (ud_eval ud term_env)
  | uc_eval (o2u ot) term_env =
      let val (ot1,_) = o_eval ot term_env
       in (o2u ot1)
      end
and ud_eval (uvar uv) term_env =
       uvar uv
  | ud_eval (uappl (ud,uc)) term_env =
       uappl ((ud_eval ud term_env),(uc_eval uc term_env));

fun nbe ty term = let val _ = setcounter 0
                   in pp_uc (uc_eval (down ty term) init_env)
                  end;
```