A Denotational Account of Untyped Normalization by Evaluation[⋆]

Andrzej Filinski[1] and Henning Korsholm Rohde[2]

[1] DIKU, University of Copenhagen, Denmark; email: andrzej@diku.dk
[2] BRICS[⋆⋆], University of Aarhus, Denmark; email: hense@brics.dk

Abstract. We show that the standard normalization-by-evaluation construction for the simply-typed $\lambda_{\beta\eta}$-calculus has a natural counterpart for the untyped λ_{β}-calculus, with the central type-indexed logical relation replaced by a "recursively defined" *invariant relation*, in the style of Pitts. In fact, the construction can be seen as generalizing a computational-adequacy argument for an untyped, call-by-name language to normalization instead of evaluation.

In the untyped setting, not all terms have normal forms, so the normalization function is necessarily partial. We establish its correctness in the senses of *soundness* (the output term, if any, is β-equivalent to the input term); *standardization* (β-equivalent terms are mapped to the same result); and *completeness* (the function is defined for all terms that do have normal forms). We also show how the semantic construction enables a simple yet formal correctness proof for the normalization algorithm, expressed as a functional program in an ML-like call-by-value language.

1 Introduction

1.1 Reduction-Based and Reduction-Free Normalization

Traditional accounts of term normalization are based on a directed notion of *reduction* (such as β-reduction), which can be applied anywhere within a term. A term is said to be a *normal form* if no reductions can be performed on it. If the reduction relation is confluent, normal forms are uniquely determined, so normalization is a (potentially partial) function on terms. Some terms (such as Ω) may not have normal forms at all; or a particular reduction strategy (such as normal-order reduction) may be required to guarantee arrival at a normal form when one exists; such a strategy is called *complete*. There is a very large body of work dealing with normalization in reduction-based settings.

However, in recent years, a rather different notion of normalization has emerged, so-called *reduction-free normalization*. As the name suggests, it is not based on a directed notion of reduction, but rather on an undirected notion of

---

[⋆] A version of this article with detailed proofs is available as a technical report [5].

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

term *equivalence*. Equivalence may be defined as simply the reflexive-transitive-symmetric closure of an existing reduction relation, but it does not have to be: any congruence relation on terms may be used. The task of normalization is then to define a *normalization function* on terms, such that the output of the function is equivalent to the input, and such that any two equivalent terms are mapped to identical outputs [3].

For some notions of equivalence (such as β-convertibility of untyped lambda-terms), it is actually impossible to define a *computable*, total normalization function with both of these properties; we must thus accept that the normalization function may be partial. However, even in that case, we can impose a completeness constraint: if we have an independent syntactic characterization of acceptable *normal forms*, we can require that the function both produce terms in this form as output, and that it be defined on all terms equivalent to a normal form.

1.2 Normalization by Evaluation

A particularly natural way of obtaining a reduction-free normalization function is known as *normalization by evaluation (NBE)*, based on the following idea: Suppose we can construct a denotational model of the term syntax (i.e., such that equivalent terms have the same denotation), with the property that a syntactic representation of the term (up to equivalence) can be be extracted from its denotation; such a model is called *residualizing.* Then the normalization function can be expressed simply as a (compositional) interpretation in the model, followed by extraction.

A priori, such a normalization function is not necessarily effectively computable. It can be given a computational interpretation if the denotational model is constructed in intuitionistic set theory [3], but this gets somewhat complicated for domain-theoretic models, especially those involving reflexive domains. In such cases, it is often easier to establish that the constructions are effective by showing that they can expressed as images of program terms in a language for which the domain-theoretic semantics is already known to be computationally adequate.

(It should be noted that the term NBE is also sometimes used for a related concept, based on reducing – usually in a compositional way – the *normalization problem*, which may in general involve open terms of higher type, to an *evaluation problem*, which involves normalization of only closed terms of base type. The required transformation is often syntactically related to the model-based construction above, but the model itself is not made explicit; and in fact, the subsequent evaluation process may still be specified entirely in terms of reductions.)

1.3 The Berger-Schwichtenberg Normalization Algorithm

Perhaps the best-known NBE algorithm is due to Berger and Schwichtenberg [2]. It finds $\beta\eta$-long normal forms of simply-typed λ-terms. We present here its outline, glossing over inessential details.

Types are of the form $\tau ::= b \mid \tau_1 \to \tau_2$. A natural set-theoretic model interprets each base type b as some set, and the function type as the set of all functions between the interpretations of the types, i.e., $[\![\tau_1 \to \tau_2]\!] = [\![\tau_1]\!] \to [\![\tau_2]\!]$. For a type assignment Γ, we also take $[\![\Gamma]\!] = \prod_{x \in \mathrm{dom}\,\Gamma} [\![\Gamma(x)]\!]$.

Let Λ be the set of syntactic λ-terms (written with explicit constructors for emphasis) over a set of variables V. For a well-typed term $\Gamma \vdash m : \tau$, we can then express its semantics $[\![m]\!] \in [\![\Gamma]\!] \to [\![\tau]\!]$ as follows:

$$\begin{aligned}
[\![\mathrm{VAR}(x)]\!]\,\rho &= \rho(x) \\
[\![\mathrm{LAM}(x^{\tau}, m_0)]\!]\,\rho &= \lambda a^{[\![\tau]\!]}.\,[\![m_0]\!]\,\rho[x \mapsto a] \\
[\![\mathrm{APP}(m_1, m_2)]\!]\,\rho &= [\![m_1]\!]\,\rho\,([\![m_2]\!]\,\rho)
\end{aligned}$$

It is easy to check that such a model is sound for conversion, i.e., that when $m \leftrightarrow_{\beta\eta} m'$, then $[\![m]\!] = [\![m']\!]$.

Consider now a model where all base types are interpreted as the set of (open) syntactic λ-terms, i.e., $[\![b]\!] = \Lambda$ for all b. In this model, we can define a pair of type-indexed function families: *reification*, $\downarrow^{\tau} : [\![\tau]\!] \to \Lambda$, and *reflection*, $\uparrow^{\tau} : \Lambda \to [\![\tau]\!]$, by mutual induction on types:

$$\begin{aligned}
\downarrow^{b} l &= l \\
\downarrow^{\tau_1 \to \tau_2} f &= \mathrm{LAM}(x^{\tau_1}, \downarrow^{\tau_2} (f(\uparrow^{\tau_1} \mathrm{VAR}(x)))) \quad \text{(where x is chosen “fresh”)} \\
\uparrow^{b} l &= l \\
\uparrow^{\tau_1 \to \tau_2} l &= \lambda a^{[\![\tau_1]\!]}.\uparrow^{\tau_2} (\mathrm{APP}(l, \downarrow^{\tau_1} a))
\end{aligned}$$

For simplicity, let us only consider normal forms of closed terms. Then reification can serve directly as an extraction function: one can check that, for a term $\vdash m : \tau$ in $\beta\eta$-long normal form, $\downarrow^{\tau} ([\![m]\!]\,\emptyset) \leftrightarrow_{\alpha} m$. Hence, by soundness of the model, for any term m' with $m' \leftrightarrow_{\beta\eta} m$, $\downarrow^{\tau} ([\![m']\!]\,\emptyset) = \downarrow^{\tau} ([\![m]\!]\,\emptyset) \leftrightarrow_{\alpha} m \leftrightarrow_{\beta\eta} m'$. Alternatively, one can show the latter property directly, for an arbitrary m'. Either way, the typical proof ultimately involves a logical-relations argument, even if this argument is pushed entirely into a standard result about the syntax (namely, that every well-typed term has a $\beta\eta$-long normal form). The latter approach, however, generalizes better, especially to systems where not all terms have normal forms.

1.4 A Tentative Algorithm for Untyped Terms

In an untyped (or, more accurately, unityped) setting, we may hope to get a residualizing model by interpreting the single type of terms as a domain $D = \Lambda + (D \to D)$. (Again, we gloss over domain-theoretic subtleties for expository purposes.) We can then define variants of reification, $\downarrow : D \to \Lambda$, and reflection, $\uparrow : \Lambda \to D$, roughly analogous to the simply-typed case:

$$\begin{aligned}
\downarrow d &= \text{case } d \text{ of } \begin{cases} in_1(l) \to l \\ in_2(f) \to \mathrm{LAM}(x, \downarrow (f(\uparrow \mathrm{VAR}(x)))) \quad (x \text{ “fresh”}) \end{cases} \\
\uparrow l &= in_1(l)
\end{aligned}$$

Note that reification is now defined by general recursion, rather than induction. We can also construct an interpretation, $[\![m]\!] \in (V \to D) \to D$, by

$$\begin{aligned}
[\![\mathrm{VAR}(x)]\!]\,\rho &= \rho(x)\\
[\![\mathrm{LAM}(x, m_0)]\!]\,\rho &= in_2(\lambda d.\,[\![m_0]\!]\,\rho[x \mapsto d])\\
[\![\mathrm{APP}(m_1, m_2)]\!]\,\rho &= \text{case } [\![m_1]\!]\,\rho \text{ of } \begin{cases} in_1(l) \to {\uparrow}(\mathrm{APP}(l, {\downarrow}([\![m_2]\!]\,\rho))) \\ in_2(f) \to f\,([\![m_2]\!]\,\rho) \end{cases}
\end{aligned}$$

Here, reflection is performed "on demand": when application needs a semantic function, but $[\![m_1]\!]\rho$ is a piece of syntax, it is reflected just enough to allow the application to be performed.

Again, it can be checked that β-convertible terms have the same denotation. It is also fairly easy to verify that, for a closed m in β-normal form, ${\downarrow}([\![m]\!]\,\emptyset) \leftrightarrow_\alpha m$. What is not obvious at all, however, is that when ${\downarrow}([\![m']\!]\,\emptyset) = m$ for a general m', then m' must be syntactically β-convertible to a normal form. Indeed, the problem is a generalization of the usual computational-adequacy problem for a denotational semantics of a functional language: if the denotation of a closed term is not $\bot$, must the term then evaluate to a value?

For a simply typed language, PCF, adequacy of the natural domain-theoretic semantics was shown by Plotkin, using a logical-relations argument [8]. Pitts showed that essentially the same argument applies to an untyped language, except that the central relation is no longer constructed by induction on types, but as a solution of a more general "relation equation"; he also showed a general method for solving such equations, yielding *invariant relations* [6].

In this paper, we first formalize the construction of the normalization function from above, addressing especially the issues of potential divergence and generation of fresh variable names (Section 2). We then show correctness of this function by a generalized computational-adequacy construction (Section 3). Finally, we show how the domain-theoretic analysis directly validates a functional program implementing the construction (Section 4).

1.5 Related Work

The closest related work to ours is probably the NBE-based (in the alternate sense) algorithm for untyped β-normalization proposed by Aehlig and Joachimski [1]. However, while the functional programs ultimately derived from the analyses are quite similar, the correctness arguments are completely different: theirs are based entirely on syntactic concepts and results from higher-order rewriting theory, rather than on the domain-theoretic constructions underlying ours. In particular, their algorithm is very explicitly reduction-based, departing from the original meaning of NBE as term extraction from a denotational model of a conversion relation.

We believe that the domain-theoretic approach enables a more direct and precise correctness proof for the normalizer, as actually implemented. In Aehlig and Joachimski's work, the abstract algorithm is expressed as a small-step operational semantics for a specialized, two-level λ-calculus with named bound variables; yet the actual normalization program is expressed as a compositional interpreter in Haskell, using de Bruijn indices for bound variables, and a reflexive type for the meanings of higher-typed terms. No connection is made to a formal semantics (operational or otherwise) of the relevant Haskell fragment. While it may well be possible to formally close this gap, it remains as a potentially major undertaking. On the other hand, formally relating the domain-theoretic constructions in the model-based normalizer to the functional terms implementing them is completely straightforward. We expect, but have not formally investigated, that Aehlig and Joachimski's interesting extensions of the basic algorithm to infinite normal forms (Böhm trees) could also be expressed naturally in the denotational setting, and be used to validate a functional program producing such normal forms lazily.

Many of the constructions in the present paper are inspired by the first author's work on type-directed partial evaluation [4]. Apart from the obvious differences arising from typed vs. untyped languages, a significant change is also that the TDPE work considered equivalence defined semantically (equality of denotations for all interpretations of "dynamic" constants), while here we consider syntactic β-convertibility. Accordingly, the central invariant relation ties denotations to syntactic terms, rather than to denotations in another semantics.

Essentially the same program as in Section 4, but expressed in FreshML, can be found in a recent paper by Shinwell et al. [9, Figure 7]. However, the focus there is on a practical application of fresh-name generation, rather than on normalization as such. Indeed, the underlying algorithm is only informally attributed to Coquand, and carries no formal correctness argument. In the present work, generation of fresh names is handled explicitly: since constructed output terms are never subsequently analyzed, using a general framework such as FreshML, or higher-order abstract syntax, is probably overkill. However, we anticipate that a different "back end" for output generation could be used, and have deliberately tried to keep the constructions and proofs modular with respect to the term-generation operations. We thus expect that essentially the same arguments – perhaps even a little simplified – could be used to verify correctness of the FreshML variant of the normalizer as well.

2 A Semantic Normalization Construction

2.1 Syntax and Semantics of the Untyped λ-Calculus

Syntax Let V be a countably infinite set of (object) variables, with x and v ranging over V. Let Λ be the set of λ-terms defined by

$$m ::= \ \mathrm{VAR}(x) \mid \mathrm{LAM}(x, m_0) \mid \mathrm{APP}(m_1, m_2)$$

The set of free variables of a term, $FV(m)$, is defined in the usual way. For any finite set of variables Δ, we write Λ^{Δ} for the set of λ-terms over Δ, i.e.,

$$\Lambda^{\Delta} = \{m \in \Lambda \mid FV(m) \subseteq \Delta\}$$

Substitutions For technical reasons, we take simultaneous (as opposed to single-variable), capture-avoiding substitution as the basic concept. Accordingly, we say that a substitution θ is a finite partial function from variables to terms. We take $FV(\theta) = \bigcup_{x \in \mathrm{dom}\,\theta} FV(\theta(x))$, and define the action of θ on a term m in the usual way, by structural induction on m:

$$\begin{aligned}
\mathrm{VAR}(x)[\theta] &= \begin{cases} \theta(x) & \text{if } x \in \mathrm{dom}\,\theta \\ \mathrm{VAR}(x) & \text{otherwise} \end{cases} \\
\mathrm{LAM}(x, m_0)[\theta] &= \mathrm{LAM}(x', m_0[\theta[x \mapsto \mathrm{VAR}(x')]]) \\
&\qquad \text{where } x' \notin FV(\theta) \cup (FV(m_0) \backslash \{x\}) \\
\mathrm{APP}(m_1, m_2)[\theta] &= \mathrm{APP}(m_1[\theta], m_2[\theta])
\end{aligned}$$

As a special case, we use the standard notation $m[m'/x]$ to mean $m[\,[x \mapsto m']\,]$. To keep the substitution operation deterministic, we assume that the x' in the LAM-clause is picked as some fixed but arbitrary function of the (finite) set of variables it needs to avoid.

Conversion and normalization We define convertibility between λ-terms, written $m \leftrightarrow m'$, by the axiom schemas for α- and β-conversion,

$$\begin{aligned}
\mathrm{LAM}(x, m) &\leftrightarrow \mathrm{LAM}(x', m[x'/x]) \quad (x' \notin FV(m) \backslash \{x\}) \\
\mathrm{APP}(\mathrm{LAM}(x, m), m') &\leftrightarrow m[m'/x]
\end{aligned}$$

together with the standard equivalence and compatibility rules, making $\leftrightarrow$ into a congruence relation on terms.
We further define *atomic* (also known as *neutral*) and *normal* forms, as follows:

$$\frac{}{\vdash_{\mathrm{at}} \mathrm{VAR}(x)} \qquad \frac{\vdash_{\mathrm{at}} m_1 \quad \vdash_{\mathrm{nf}} m_2}{\vdash_{\mathrm{at}} \mathrm{APP}(m_1, m_2)} \qquad \frac{\vdash_{\mathrm{at}} m}{\vdash_{\mathrm{nf}} m} \qquad \frac{\vdash_{\mathrm{nf}} m_0}{\vdash_{\mathrm{nf}} \mathrm{LAM}(x, m_0)}$$

We then expect a normalization function on terms to satisfy that the output, if any, is in normal form and convertible to the input (soundness); convertible terms either give the same output, or neither one does (standardization); and if a term has a normal form at all, the normalization function will return one (completeness).

Semantics A natural way of defining a denotational model of convertibility is in terms of a reflexive pointed cpo D. Reflexivity means that the continuous-function space $[D \to D]$ is a retract of D, i.e., that there exist continuous functions

$$\phi : [D \to D] \to D \quad \text{and} \quad \psi : D \to [D \to D]\,,$$

such that $\psi \circ \phi = id_{[D \to D]}$. The induced interpretation, $[\![m]\!] \in [[V \to D] \to D]$, is then:

$$\begin{aligned}
[\![\mathrm{VAR}(x)]\!]\,\rho &= \rho(x) \\
[\![\mathrm{LAM}(x, m_0)]\!]\,\rho &= \phi(\lambda d^D.\, [\![m_0]\!]\,\rho[x \mapsto d]) \\
[\![\mathrm{APP}(m_1, m_2)]\!]\,\rho &= \psi([\![m_1]\!]\,\rho)\,([\![m_2]\!]\,\rho)
\end{aligned}$$

Lemma 1. *The interpretation has two expectable properties:*

a. If $\forall x \in FV(m).\, \rho(x) = \rho'(x)$*, then* $[\![m]\!]\,\rho = [\![m]\!]\,\rho'$.
b. Let $\theta = [x_1 \mapsto m_1, \ldots, x_n \mapsto m_n]$ *be a substitution.*
Then $[\![m[\theta]]\!]\,\rho = [\![m]\!]\,\rho[x_1 \mapsto [\![m_1]\!]\,\rho, \ldots, x_n \mapsto [\![m_n]\!]\,\rho]$.

Proof. Part (a) is a straightforward induction on the structure of m. Part (b) follows by induction on the structure of m, using part (a) in the LAM-case.

Lemma 2 (model soundness). *If* $m \leftrightarrow m'$ *then* $[\![m]\!] = [\![m']\!]$

Proof. By induction on the derivation of $m \leftrightarrow m'$, using Lemma 1 for α- and β-conversion, and using that $\psi \circ \phi = id_{[D \to D]}$ for β-conversion.

2.2 Output-Term Generation

We want to account rigorously for the generation of fresh names, and do so in a modular manner. We will therefore construct a set $\widehat{\varLambda}$ (dependent on the name generation scheme) with elements denoted by l, together with *wrapper* functions,

$$\widehat{\mathrm{VAR}} : V \to \widehat{\varLambda}, \quad \widehat{\mathrm{LAM}} : [V \to \widehat{\varLambda}] \to \widehat{\varLambda}, \quad \widehat{\mathrm{APP}} : \widehat{\varLambda} \times \widehat{\varLambda} \to \widehat{\varLambda}$$

where, in particular, $\widehat{\mathrm{LAM}}$ provides a fresh name to be used in constructing the body of the λ-abstraction.

Let $\mathcal{N}$ be a set (discrete cpo) containing at least the natural numbers, with an operation $\cdot + 1 : \mathcal{N} \to \mathcal{N}$, agreeing with the successor operation on naturals. Let $\{g_0, g_1, ...\}$ be a countably infinite subset of V, such that $g_i = g_j$ implies $i = j$, and let $gen : \mathcal{N} \to V$ be such that $gen(n) = g_n$ when $n \in \mathbb{N}$.

We write $\lfloor \cdot \rfloor$ for the inclusion from A to $A_\perp$; and for $f : A \to B$ with B pointed, we write $\cdot \star f$ for f's strict extension to $A_\perp$, i.e., $\perp \star f = \perp_B$ and $\lfloor a \rfloor \star f = f\, a$. We then take $\widehat{\varLambda} = [\mathcal{N} \to \varLambda_\perp]$ and define wrapper functions for constructing λ-terms using de Bruijn-level (not -index!) naming as follows:

$$\begin{aligned}
\widehat{\mathrm{VAR}}(v) &= \lambda n^{\mathcal{N}}.\, \lfloor \mathrm{VAR}(v) \rfloor \\
\widehat{\mathrm{LAM}}(f) &= \lambda n^{\mathcal{N}}.\, f\ gen(n)\ (n+1) \star \lambda m_0^{\varLambda}.\, \lfloor \mathrm{LAM}(gen(n), m_0) \rfloor \\
\widehat{\mathrm{APP}}(l_1, l_2) &= \lambda n^{\mathcal{N}}.\, l_1\, n \star \lambda m_1^{\varLambda}.\, l_2\, n \star \lambda m_2^{\varLambda}.\, \lfloor \mathrm{APP}(m_1, m_2) \rfloor
\end{aligned}$$

Note 1. If we took freshness as a primitive concept, like in FreshML, we could simply use $\widehat{\varLambda} = \varLambda_\perp$; $\widehat{\mathrm{VAR}}(v) = \lfloor \mathrm{VAR}(v) \rfloor$; $\widehat{\mathrm{LAM}}(f) = f\, x \star \lambda m_0.\, \lfloor \mathrm{LAM}(x, m_0) \rfloor$, with x *fresh* for f; and $\widehat{\mathrm{APP}}(l_1, l_2) = l_1 \star \lambda m_1.\, l_2 \star \lambda m_2.\, \lfloor \mathrm{APP}(m_1, m_2) \rfloor$.

2.3 A Residualizing Model

From standard domain-theoretic results (e.g., [6]), we know that there exists a pointed cpo D_r, together with an isomorphism

$$i : D_r \xrightarrow{\simeq} (\widehat{\varLambda} + [D_r \to D_r])_\perp$$

Moreover, this solution is a so-called *minimal invariant*, which we will need in the next section.

We first define the reification function $\uparrow\, : \widehat{\Lambda} \to D_r$ and reflection function $\downarrow\, : D_r \to \widehat{\Lambda}$, as follows:

$$
\begin{aligned}
\downarrow d &= \text{case } i(d) \text{ of} \begin{cases} \lfloor in_1(l) \rfloor & \to l \\ \lfloor in_2(f) \rfloor & \to \widehat{\mathrm{LAM}}(\lambda x^V. \downarrow (f(\uparrow \widehat{\mathrm{VAR}}(x)))) \\ \quad \bot & \to \bot_{\widehat{\Lambda}} \end{cases} \\
\uparrow l &= i^{-1}(\lfloor in_1(l) \rfloor)
\end{aligned}
$$

where the recursive definition of $\downarrow$ is interpreted in the usual least-fixed-point sense. Using these, we construct appropriate functions $\phi_r : [D_r \to D_r] \to D_r$ and $\psi_r : D_r \to [D_r \to D_r]$:

$$
\begin{aligned}
\phi_r(f) &= i^{-1}(\lfloor in_2(f) \rfloor) \\
\psi_r(d) &= \text{case } i(d) \text{ of} \begin{cases} \lfloor in_1(l) \rfloor & \to \lambda d'^{D_r}. \uparrow \widehat{\mathrm{APP}}(l, \downarrow d') \\ \lfloor in_2(f) \rfloor & \to f \\ \quad \bot & \to \bot_{[D_r \to D_r]} \end{cases}
\end{aligned}
$$

Clearly, we have that $\psi_r \circ \phi_r = id_{[D_r \to D_r]}$, since i was an isomorphism. The induced interpretation is denoted by $[\![\cdot]\!]_r$. We can now define a putative normalization function:

Definition 1. *For any Δ, let $\sharp\Delta = \max(\{n+1 \mid g_n \in \Delta\} \cup \{0\})$ (i.e., the least n such that $\forall n' \geq n.\, g_{n'} \notin \Delta$). We then define the function* $\mathrm{norm}_\Delta : \Lambda^\Delta \to \Lambda_\bot$ *by*

$$\mathrm{norm}_\Delta(m) = \downarrow ([\![m]\!]_r\, (\lambda x^V. \uparrow \widehat{\mathrm{VAR}}(x)))\ \sharp\Delta$$

In particular, when Δ is disjoint from the set of g_i-names (so $\sharp\Delta = 0$), we write just norm *for* norm_Δ.

3 Correctness of the Construction

3.1 Correctness of the Wrappers

Let $s \in \{\mathrm{at}, \mathrm{nf}\}$ be a syntactic-form designator. We first define a quaternary relation, $l \lessapprox_s^\Delta m$, expressing that if l represents a term at all, then that term only has free variables in Δ, is of the syntactic form s, and is convertible to m:

Definition 2. *For $l \in \widehat{\Lambda}$ and $m \in \Lambda^\Delta$, we then define the relation $\lessapprox$ by*

$$l \lessapprox_s^\Delta m \quad \textit{iff} \quad \forall n \geq \sharp\Delta, m' \in \Lambda.\, l\, n = \lfloor m' \rfloor \Rightarrow m' \in \Lambda^\Delta \wedge \vdash_s m' \wedge m' \leftrightarrow m$$

Lemma 3. *For fixed Δ, s, and m, the predicate $P = \{l \mid l \lessapprox_s^\Delta m\}$ is pointed (i.e., $\bot_{\widehat{\Lambda}} \in P$) and inclusive (i.e., closed under limits of ω-chains).*

Proof. Straightforward, noting that $\lessapprox$ is expressed using intersection, inverse image, and a (necessarily inclusive) predicate on the flat domain $\Lambda_\bot$.

Lemma 4. *The representation relation is closed under weakening and conversion:*

a. If $l \lessapprox_s^{\Delta} m$ *and* $\Delta \subseteq \Delta'$*, then also* $l \lessapprox_s^{\Delta'} m$*.*
b. If $l \lessapprox_s^{\Delta} m$ *and* $m' \in \Lambda^{\Delta}$ *with* $m \leftrightarrow m'$*, then also* $l \lessapprox_s^{\Delta} m'$*.*

Proof. Both parts are immediate from the definition.

Lemma 5. *Representations of terms behave much like the terms themselves:*

a. If $v \in \Delta$ *then* $\widehat{\mathrm{VAR}}(v) \lessapprox_{\mathrm{at}}^{\Delta} \mathrm{VAR}(v)$*.*
b. If $l_1 \lessapprox_{\mathrm{at}}^{\Delta} m_1$ *and* $l_2 \lessapprox_{\mathrm{nf}}^{\Delta} m_2$*, then* $\widehat{\mathrm{APP}}(l_1, l_2) \lessapprox_{\mathrm{at}}^{\Delta} \mathrm{APP}(m_1, m_2)$*.*
c. If $l \lessapprox_{\mathrm{at}}^{\Delta} m$*, then also* $l \lessapprox_{\mathrm{nf}}^{\Delta} m$*.*
d. Let $f \in [V \to \widehat{\Lambda}]$ *and* $m \in \Lambda^{\Delta \cup \{x\}}$*. If* $\forall v \notin \Delta . f v \lessapprox_{\mathrm{nf}}^{\Delta \cup \{v\}} m[\mathrm{VAR}(v)/x]$*, then* $\widehat{\mathrm{LAM}}(f) \lessapprox_{\mathrm{nf}}^{\Delta} \mathrm{LAM}(x, m)$*.*

Proof. All parts are relatively straightforward. (b) and (d) exploit that $\leftrightarrow$ is a congruence relation. For (d), the assumption about m's free variables is also essential.

3.2 Adequacy of the Residualizing Model

To construct the central relation between denotations and terms, we first state an abstract version of a result due to Pitts [6]:

Theorem 1 (existence of invariant relations). *Let* A *be a cpo, and let* $i : D \xrightarrow{\simeq} (A + [D \to D])_\perp$ *be a minimal-invariant solution of the domain equation* $X \cong (A + [X \to X])_\perp$*. Let* T *be a set, and let predicates* $P_1 \subseteq A \times T$*,* $P_2 \subseteq T$*, and* $P_3 \subseteq T \times T \times T$ *be given, such that* $\{a \mid P_1(a, t)\}$ *is inclusive for every* $t \in T$*. Then there exists a relation* $\lhd \subseteq D \times T$*, with* $\{d \mid d \lhd t\}$ *inclusive for every* $t \in T$*, and such that, for all* $d \in D$ *and* $t \in T$*:*

$$
\begin{aligned}
d \lhd t \textit{ iff } \; & i(d) = \perp \\
& \textit{or } \exists a.\, i(d) = \lfloor in_1(a) \rfloor \wedge P_1(a, t) \\
& \textit{or } \exists f.\, i(d) = \lfloor in_2(f) \rfloor \wedge P_2(t) \wedge \\
& \qquad \forall d' \in D; t', t'' \in T.\ P_3(t, t', t'') \wedge d' \lhd t' \Rightarrow f(d') \lhd t''.
\end{aligned}
$$

Proof. The proof proceeds exactly as in Pitts's paper, with the following minor refinements: First, the cpo A can be arbitrary (not necessarily discrete), as long as the relation P_1 is inclusive. Also, when P_2 is an existential proposition, the witness need not be unique (such as the result of a deterministic evaluation), as long as the choice of witness does not affect P_3.

We can then establish the existence of a Kripke-style invariant relation, using sets of variables as worlds:

Lemma 6. *There exists a relation $\lesssim$ such that for all Δ, $d \in D_r$ and $m \in \Lambda^{\Delta}$,*

$$\begin{array}{l}
d \lesssim^{\Delta} m \textit{ iff } i(d) = \bot \\
\qquad\qquad \textit{or } \exists l.\, i(d) = \lfloor in_1(l) \rfloor \wedge l \lessapprox^{\Delta}_{\mathrm{at}} m \\
\qquad\qquad \textit{or } \exists f.\, i(d) = \lfloor in_2(f) \rfloor \wedge (\exists x \in V, m_0 \in \Lambda^{\Delta \cup \{x\}}.\mathrm{LAM}(x, m_0) \leftrightarrow m) \\
\qquad\qquad\qquad \wedge\, \forall \Delta' \supseteq \Delta, d' \in D_r, m' \in \Lambda^{\Delta'}, m_1 \in \Lambda^{\Delta'}. \\
\qquad\qquad\qquad\qquad m \leftrightarrow m_1 \wedge d' \lesssim^{\Delta'} m' \Rightarrow f(d') \lesssim^{\Delta'} \mathrm{APP}(m_1, m')
\end{array}$$

Proof. By Theorem 1, taking $A = \widehat{\Lambda}$ and $T = \{(\Delta, m) \mid \Delta \subseteq_{\mathrm{fin}} V \wedge m \in \Lambda^{\Delta}\}$, with the predicates chosen as

$$\begin{array}{rcl}
P_1 & = & \{(l, (\Delta, m)) \mid l \lessapprox^{\Delta}_{\mathrm{at}} m\} \\
P_2 & = & \{(\Delta, m) \mid \exists x \in V, m_0 \in \Lambda^{\Delta \cup \{x\}}.\mathrm{LAM}(x, m_0) \leftrightarrow m\} \\
P_3 & = & \{((\Delta, m), (\Delta', m'), (\Delta'', m'')) \mid \\
 & & \qquad \Delta \subseteq \Delta' = \Delta'' \wedge \exists m_1 \in \Lambda^{\Delta'}.m \leftrightarrow m_1 \wedge m'' = \mathrm{APP}(m_1, m')\}
\end{array}$$

using the equivalence $[\forall x.(\exists y.P(x,y)) \Rightarrow Q(x)] \Leftrightarrow [\forall x.\forall y.P(x,y) \Rightarrow Q(x)]$. P_1 is inclusive in its first argument by Lemma 3. We write $d \lesssim^{\Delta} m$ instead of $d \lhd (\Delta, m)$.

Lemma 7. *The relation $\lesssim$ shares two key properties with $\lessapprox$:*

a. If $d \lesssim^{\Delta} m$ and $\Delta \subseteq \Delta'$, then also $d \lesssim^{\Delta'} m$.
b. If $d \lesssim^{\Delta} m$ and $m' \in \Lambda^{\Delta}$ with $m \leftrightarrow m'$, then also $d \lesssim^{\Delta} m'$.

Proof. Both parts are straightforward, given Lemma 4, and noting the transitivity of $\subseteq$ and $\leftrightarrow$.

The following two lemmas will combine to establish adequacy of our semantics:

Lemma 8. *For all $l \in \widehat{\Lambda}$, $d \in D_r$, and $m \in \Lambda^{\Delta}$,*

a. If $l \lessapprox^{\Delta}_{\mathrm{at}} m$ then $\uparrow l \lesssim^{\Delta} m$
b. If $d \lesssim^{\Delta} m$ then $\downarrow d \lessapprox^{\Delta}_{\mathrm{nf}} m$

Proof. Part (a) follows immediately from the definition of $\uparrow$. Part (b) exploits $\downarrow$'s definition as a least fixed point and proceeds by fixed-point induction on the pointed and inclusive (by Lemma 3) predicate

$$R = \{\varphi \in [D_r \to \widehat{\Lambda}] \mid \forall d, \Delta, m \in \Lambda^{\Delta}.\, d \lesssim^{\Delta} m \Rightarrow \varphi(d) \lessapprox^{\Delta}_{\mathrm{nf}} m\}$$

The verification uses the properties of $\widehat{\Lambda}$-representations (Lemma 5(a,c,d)), and that both $\lessapprox$ and $\lesssim$ are closed under conversion (Lemmas 4(b) and 7(b)).

Lemma 9. *Let $m \in \Lambda^{\Gamma}$, and for all $x \in \Gamma$, let $\theta(x) \in \Lambda^{\Delta}$ (in particular, $\Gamma \subseteq \mathrm{dom}\,\theta$). If $\forall x \in \Gamma.\, \rho(x) \lesssim^{\Delta} \theta(x)$ then $[\![m]\!]_r\, \rho \lesssim^{\Delta} m[\theta]$.*

Proof. By structural induction on m. The case for variables is immediate. For abstractions, like in a standard Kripke-logical-relations proof, monotonicity of $\lesssim$ (Lemma 7(a)) ensures that the environment and substitution remain related in the later world Δ'; also, closure under conversion (Lemma 7(b)) in particular implies closure under β-expansion. Both parts of Lemma 8, as well as Lemma 5(b), are used in the non-standard case for applications.

3.3 Correctness of the Normalization Function

Definition 3. *The predicate* $\mathrm{tot}(\cdot) \subseteq \widehat{\Lambda}$ *is given by* $\mathrm{tot}(l) \Leftrightarrow \forall n \in \mathbb{N}.\, l\, n \neq \bot$.

Lemma 10. *The following properties hold of the wrapper functions:*

a. For all $v \in V$, $\mathrm{tot}(\widehat{\mathrm{VAR}}(v))$.
b. If for all $v \in V$. $\mathrm{tot}(f\, v)$ *then* $\mathrm{tot}(\widehat{\mathrm{LAM}}(f))$.
c. If $\mathrm{tot}(l_1)$ *and* $\mathrm{tot}(l_2)$ *then* $\mathrm{tot}(\widehat{\mathrm{APP}}(l_1, l_2))$.

Proof. Straightforward verification in each case.

Lemma 11. *For all* $m \in \Lambda$ *and* $\rho \in [V \to D_r]$ *such that for all* $x \in FV(m)$, *there exists an* l *with* $\rho(x) = \uparrow l$ *and* $\mathrm{tot}(l)$,

a. If $\vdash_{\mathrm{at}} m$ *then* $\exists l \in \widehat{\Lambda}.\, [\![m]\!]_r\, \rho = \uparrow l \wedge \mathrm{tot}(l)$.
b. If $\vdash_{\mathrm{nf}} m$ *then* $\mathrm{tot}(\downarrow ([\![m]\!]_r\, \rho))$.

Proof. By simultaneous rule induction on $\vdash_{\mathrm{at}} \cdot$ and $\vdash_{\mathrm{nf}} \cdot$, relying on Lemma 10 for the totality properties of the wrappers.

Theorem 2 (semantic correctness). norm_Δ *from Definition 1 is a normalization function on* Λ^Δ, *i.e.,*

a. (soundness) If $\mathrm{norm}_\Delta(m) = \lfloor m' \rfloor$ *then* $m' \in \Lambda^\Delta$, $\vdash_{\mathrm{nf}} m'$, *and* $m \leftrightarrow m'$.
b. (standardization) If $m \leftrightarrow m'$ *then* $\mathrm{norm}_\Delta(m) = \mathrm{norm}_\Delta(m')$.
c. (completeness) If $m \leftrightarrow m'$ *with* $\vdash_{\mathrm{nf}} m'$ *then* $\mathrm{norm}_\Delta(m) \neq \bot$.

Proof. *(Soundness)* Let θ_0 be the substitution mapping every x in Δ to $\mathrm{VAR}(x)$, and $\rho_0 = \lambda x^V . \uparrow \widehat{\mathrm{VAR}}(x)$. By Lemma 5(a), for every $x \in \Delta$, $\widehat{\mathrm{VAR}}(x) \lessapprox^{\Delta}_{\mathrm{at}} \mathrm{VAR}(x) = \theta_0(x)$, and hence by Lemma 8(a), $\rho_0(x) \lesssim^{\Delta} \theta_0(x)$. By Lemma 9, we then get that $[\![m]\!]_r\, \rho_0 \lesssim^{\Delta} m[\theta_0] \leftrightarrow m$, and therefore, by Lemma 8(b), $\downarrow ([\![m]\!]_r\, \rho_0) \lessapprox^{\Delta}_{\mathrm{nf}} m$. Assume now that $\mathrm{norm}_\Delta(m) = \lfloor m' \rfloor$. Taking $n = \sharp\Delta$ in Definition 2, we can then immediately read off that m' has the required properties.

(Standardization) This follows directly from model soundness (Lemma 2), since the residualizing model is indeed a model.

(Completeness) Using Lemma 10(a), we see that ρ_0 satisfies the condition on ρ in Lemma 11. Hence, by part (b) of the latter lemma and Definition 3, $\mathrm{norm}_\Delta(m') \neq \bot$. The desired result then follows from (standardization).

4 An Implementation of the Construction

4.1 Syntax and Semantics of an ML-like Call-by-Value Language

The language is a small fragment of Standard ML where, to sidestep inessential bookkeeping, we have hard-coded the inductive representation of λ-terms,

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

as an additional base type of the language, and simply taken the value sets underlying `string` and `term` to be the sets V and Λ, respectively.

Syntax The fragment is restricted to a single recursive datatype declaration,

$$\texttt{datatype}\ \textit{dt}\ \texttt{=}\ \textit{In}_1\ \texttt{of}\ \tau^1\ \texttt{|}\ \cdots\ \texttt{|}\ \textit{In}_k\ \texttt{of}\ \tau^k$$

where types are given by the grammar

$$\tau ::= \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{term} \mid \tau_1 \,\texttt{->}\, \tau_2 \mid \textit{dt}$$

The syntax of ML expressions is then

$$\begin{aligned} e ::= \ & x \mid \underline{n} \mid \texttt{"}v\texttt{"} \mid \texttt{()} \mid e_1 \,\texttt{+}\, e_2 \mid e_1 \,\texttt{=}\, e_2 \mid \texttt{"g"\^{}Int.toString}\ e \mid \\ & \texttt{fn () =>}\ e \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e_1\ e_2 \mid \texttt{VAR(}e\texttt{)} \mid \texttt{LAM(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{APP(}e_1\texttt{,}e_2\texttt{)} \mid \\ & \texttt{case}\ e\ \texttt{of VAR}\ x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{LAM(}x_2\texttt{,}x_2'\texttt{)}\ \texttt{=>}\ e_2\ \texttt{|}\ \texttt{APP(}x_3\texttt{,}x_3'\texttt{)}\ \texttt{=>}\ e_3 \mid \\ & \textit{In}_i\texttt{(}e\texttt{)} \mid \texttt{case}\ e\ \texttt{of}\ \textit{In}_1\ x_1\ \texttt{=>}\ e_1 \texttt{|}\ \cdots\ \texttt{|}\ \textit{In}_k\ x_k\ \texttt{=>}\ e_k \mid \\ & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{let fun}\ f\ \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \end{aligned}$$

where x and f range over ML variable names.

Typing We only consider well-typed ML expressions, as captured by the judgement $x_1{:}\tau_1, ..., x_n{:}\tau_n \vdash e : \tau$, asserting that e is of type τ, with free variables $x_1, ..., x_n$ of types $\tau_1, ..., \tau_n$. It is defined in the usual way by inference rules.

Operational semantics A *complete program* is a closed expression of type $\tau_1 \texttt{->} \tau_2$, where τ_1 and τ_2 are ground types (i.e., not containing `->` or $\textit{dt}$). For such types, let C_τ denote the set of canonical values underlying τ, e.g., $C_{\texttt{int}} = \mathbb{Z}$.

For a complete program $e : \tau_1 \,\texttt{->}\, \tau_2$, we can construct a computable partial function $\mathrm{run}_e : C_{\tau_1} \rightharpoonup C_{\tau_2}$, e.g., by

$$\mathrm{run}_e(c_1) = c_2 \ \text{ iff } \ (e\ \underline{c_1}) \Downarrow \underline{c_2}.$$

where $\Downarrow$ is the usual big-step operational semantics of expressions, and $\underline{c}$ denotes the syntactic representation of the value c.

Denotational Semantics For the meaning of ML types, we take

$$\begin{array}{cccc} \llbracket\texttt{unit}\rrbracket^{\mathrm{ml}} = \mathbf{1} = \{*\} & \llbracket\texttt{int}\rrbracket^{\mathrm{ml}} = \mathbb{Z} & \llbracket\texttt{bool}\rrbracket^{\mathrm{ml}} = \mathbb{B} & \llbracket\texttt{string}\rrbracket^{\mathrm{ml}} = V \\ \llbracket\texttt{term}\rrbracket^{\mathrm{ml}} = \mathit{\Lambda} & \llbracket\tau_1 \,\texttt{->}\, \tau_2\rrbracket^{\mathrm{ml}} = [\llbracket\tau_1\rrbracket^{\mathrm{ml}} \to \llbracket\tau_2\rrbracket^{\mathrm{ml}}_{\perp}] & \llbracket\textit{dt}\rrbracket^{\mathrm{ml}} = S & \end{array}$$

where $i_S : S \overset{\simeq}{\Rightarrow} \llbracket\tau^1\rrbracket^{\mathrm{ml}} + \cdots + \llbracket\tau^k\rrbracket^{\mathrm{ml}}$ is a minimal-invariant solution to the evident predomain equation. We write $in_i : \llbracket\tau^i\rrbracket^{\mathrm{ml}} \to \llbracket\tau^1\rrbracket^{\mathrm{ml}} + \cdots + \llbracket\tau^k\rrbracket^{\mathrm{ml}}$ for the injection functions.

The meaning of ML terms is defined by induction on the typing derivation; for conciseness we write only the terms. The semantics is structured such that if $\mathit{\Gamma} \vdash e : \tau$ and for all $(x : \tau') \in \mathit{\Gamma}$, $\xi(x) \in \llbracket\tau'\rrbracket^{\mathrm{ml}}$, then $\llbracket e\rrbracket^{\mathrm{ml}}\xi \in \llbracket\tau\rrbracket^{\mathrm{ml}}_{\perp}$. In particular, the semantics of $\textit{dt}$-constructors and recursive function definitions are given by:

$$\begin{aligned} & \llbracket \textit{In}_i\texttt{(}e\texttt{)}\rrbracket^{\mathrm{ml}}\,\xi \ = \ \llbracket e\rrbracket^{\mathrm{ml}}\,\xi \star \lambda a^{\llbracket\tau^i\rrbracket^{\mathrm{ml}}}.\,\lfloor i_S^{-1}(in_i(a))\rfloor \\ & \llbracket \texttt{let fun}\ f\ \texttt{(}x\texttt{:}\tau_1\texttt{):}\tau_2\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}\rrbracket^{\mathrm{ml}}\,\xi \ = \\ & \quad \llbracket e_2\rrbracket^{\mathrm{ml}}\,\xi[f \mapsto \mathrm{fix}(\lambda\theta^{[\llbracket\tau_1\rrbracket^{\mathrm{ml}} \to \llbracket\tau_2\rrbracket^{\mathrm{ml}}_{\perp}]}.\lambda a^{\llbracket\tau_1\rrbracket^{\mathrm{ml}}}.\llbracket e_1\rrbracket^{\mathrm{ml}}\,\xi[f \mapsto \theta, x \mapsto a])] \end{aligned}$$

For notational convenience in the following, we will assume that all function names f in the program are distinct. We can then unambiguously use Θ_f to refer to the semantic function whose fixed point f is mapped to in the environment of the `let`-body, and $\theta_f = \mathrm{fix}(\Theta_f)$.

Theorem 3 (computational adequacy for ML). *For a complete ML program e, $run_e(c_1) = c_2$ iff $[\![e]\!]^{\mathrm{ml}}\,\emptyset \star \lambda f.\, f(c_1) = \lfloor c_2 \rfloor$.*

Proof. Modulo trivial syntactic differences, and an equivalent formulation of the semantics in terms of strict functions between pointed cpos, rather than general ones between cpos, this is shown in, e.g., [7, Section 5]. The primary difficulty is, of course, the definition of the logical relation at type $\boldsymbol{dt}$, which is again achieved by exploiting the minimal-invariant property of S.

4.2 The Normalization Algorithm

The concrete representation of the normalization algorithm, with many of the auxiliary definitions inlined, is shown in Fig. 1. We have instantiated $\boldsymbol{dt}$ as the type `sem`, with two constructors $\boldsymbol{In}_1 =$ `TM` and $\boldsymbol{In}_2 =$ `FUN`. It is easy to check that the top-level expression, $NORM$: `term -> term`, is a well-typed complete program in our sense.

Since ML is a call-by-value language, we must simulate the implicit call-by-name nature of the residualizing semantics using thunking. We have defined `sem` so that $[\![\mathtt{sem}]\!]^{\mathrm{ml}}_{\bot} \cong D_r$; then semantic functions with codomain D_r can be represented directly as ML functions into `sem`, while functions with domain D_r are represented with source type `unit -> sem`. As a further optimization, the *strict* function $\downarrow\, : D_r \to \widehat{\Lambda}$ is represented as simply a function from `sem`.

Let us now properly relate the abstract and concrete constructions. To get a perfect isomorphism between term families and their implementation, we choose $\mathcal{N} = \mathbb{Z}$, with $gen(n) =$ `"g`$\underline{n}$`"`, e.g., $gen(13) =$ `"g13"`. Let i_D denote the isomorphism $i : D_r \overset{\cong}{\Rightarrow} ([\mathbb{Z} \to \Lambda_\bot] + [D_r \to D_r])_\bot$ from before. We now also have $i_S : S \overset{\cong}{\Rightarrow} [\mathbb{Z} \to \Lambda_\bot] + [[\mathbf{1} \to S_\bot] \to S_\bot]$.

Lemma 12. *There exists an isomorphism $i_{DS} : D_r \overset{\cong}{\Rightarrow} S_\bot$, satisfying*

a. For all $l \in \widehat{\Lambda}$, $i_{DS}(i_D^{-1}(\lfloor in_1(l) \rfloor)) = \lfloor i_S^{-1}(in_1(l)) \rfloor$.
b. For all $f \in [D_r \to D_r]$,
$$i_{DS}(i_D^{-1}(\lfloor in_2(f) \rfloor)) = \lfloor i_S^{-1}(in_2(\lambda t^{\mathbf{1} \to S_\bot}.\, i_{DS}(f(i_{DS}^{-1}(t\,*))))) \rfloor.$$
c. $i_{DS}(i_D^{-1}(\bot_{D_r})) = \bot_{S_\bot}$

Proof. The strict functions (i_{DS}, i_{DS}^{-1}) are constructed in the natural way by mutual recursion. That they are actually inverses, and satisfy equations (a) and (b), follows from the minimal-invariant properties of D_r and S.

We can also state three lemmas, relating the central domain-theoretic functions to the denotations of their syntactic counterparts:

Lemma 13. *For all $d \in D_r$ and $n \in \mathbb{Z}$, $\downarrow d\, n = i_{DS}(d) \star \lambda s^S.\, \theta_{\mathtt{down}}\ s \star \lambda l^{\widehat{\Lambda}}.\, l\, n$.*

```
datatype term = VAR of string | LAM of string*term | APP of term*term
datatype sem = TM of int -> term | FUN of (unit -> sem) -> sem;

let fun down (s:sem):int->term = fn n =>
     (case s of
        TM l => l n
      | FUN f => LAM("g"^Int.toString n,
          down (f (fn () => TM(fn n' => VAR("g"^Int.toString n)))) (n+1)))
in let fun eval (m:term):(string->sem)->sem = fn p =>
        (case m of
           VAR x => p x
         | LAM(x,m0) => FUN(fn d => eval m0
                              (fn x' => if x = x' then d () else p x'))
         | APP(m1,m2) => (case (eval m1 p) of
                            TM l => TM(fn n => APP(l n,down (eval m2 p) n))
                          | FUN f => f (fn () => eval m2 p)))
   in let fun norm (m:term):term =
            down (eval m (fn x => TM(fn n => VAR(x)))) 0
      in norm end end end
```

Fig. 1. The normalization algorithm, *NORM*, in a fragment of ML

Proof. By simultaneous fixed-point induction wrt. the relation $R = \{(\varphi, \theta) \in [D_r \to \widehat{\Lambda}] \times [S \to \widehat{\Lambda}_\perp] \mid \forall d \in D_r, n \in \mathbb{Z}.\, \varphi\ d\ n = i_{DS}(d) \star \lambda s^S.\theta\ s \star \lambda l^{\widehat{\Lambda}}.l\ n\}$. The inductive step proceeds by analysis of the cases for $i_D(d)$; they all follow straightforwardly, using Lemma 12.

Lemma 14. *For all* $m \in \Lambda$, $\rho \in [V \to D_r]$, *and* $\zeta \in [V \to S_\perp]$, *such that* $\forall x \in FV(m).\, i_{DS}(\rho(x)) = \zeta(x)$, $i_{DS}([\![m]\!]_r\,\rho) = \theta_{\mathtt{eval}}\, m \star \lambda g.\, g\, \zeta$.

Proof. By structural induction on m, using the fixed-point equation for $\mathrm{fix}(\Theta_{\mathtt{eval}})$ in the inductive steps, Lemma 12 throughout, and Lemma 13 in one of the subcases for application.

Lemma 15. *For all* $m \in \Lambda$, $\mathrm{norm}(m) = \theta_{\mathtt{norm}}\ m$.

Proof. Follows easily from the definition of $\theta_{\mathtt{norm}}$, using Lemmas 12, 13 and 14.

Theorem 4 (implementation correctness). *The program NORM satisfies that* $\mathrm{run}_{NORM}(m) = m' \Leftrightarrow \mathrm{norm}(m) = \lfloor m' \rfloor$. *That is, NORM computes the normalization function for all* λ*-terms without free occurrences of* $\mathrm{g}\underline{n}$*-variables (including, in particular, all closed terms).*

Proof. A direct consequence of Lemma 15 and Theorem 3.

5 Conclusions and Perspectives

We have presented a domain-theoretic analysis of a normalization-by-evaluation construction for untyped λ-terms. Compared to the typed case, the main difference is a change from induction on types to general recursion, both for function definitions and for the domains and relations on them. That the correctness proof has a generalized computational-adequacy result at its core, further strengthens the connection between normalization and evaluation. Moreover, the algorithmic content of the construction corresponds very directly to a simple functional program, enabling a precise verification of the normalizer as actually implemented.

There are several possible directions in which to extend the present work. Some were already mentioned in Section 1.5, such as generalizations of the algorithm to Böhm trees. It should also be possible to extend the language and notion of normalization with interpreted constants in a suitable sense. But already the current results indicate that the fundamental ideas of NBE are not incompatible with general recursive types. Thus, reduction-free normalization may provide a complementary view of other equational systems that are currently analyzed using exclusively reduction-based methods. It might even be possible to find unified formulations of rewriting-theoretic and model-theoretic normalization results about particular such systems.

Acknowledgment The authors wish to thank Olivier Danvy and the FOSSACS'04 reviewers for their insightful comments.

References

1. Klaus Aehlig and Felix Joachimski. Operational aspects of untyped normalization by evaluation. *Mathematical Structures in Computer Science*, 2004(?). To appear; available from `http://www.mathematik.uni-muenchen.de/~aehlig/pub/03-nbe.ps`.
2. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991.
3. Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.
4. Andrzej Filinski. A semantic account of type-directed partial evaluation. In G. Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 378–395, Paris, France, September 1999. Springer-Verlag.
5. Andrzej Filinski and Henning Korsholm Rohde. A denotational account of untyped normalization by evaluation (extended version). BRICS Report RS-03-40, University of Aarhus, Denmark, December 2003. Available from `http://www.brics.dk/RS/03/40/`.
6. Andrew M. Pitts. Computational adequacy via 'mixed' inductive definitions. In *Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 72–82. Springer-Verlag, April 1993.
7. Andrew M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, June 1996.
8. Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.
9. Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 263–274. ACM Press, Uppsala, Sweden, August 2003.