Under consideration for publication in Math. Struct. in Comp. Science

Operational Aspects of Untyped Normalization by Evaluation

KLAUS AEHLIG[†] and FELIX JOACHIMSKI[†]

Mathematisches Institut der Ludwig-Maximilians-Universität München
Theresienstrasse 39, 80333 München, Germany

Received 15 April 2003

A purely syntactic and untyped variant of Normalization by Evaluation for the λ-calculus is presented in the framework of a two-level λ-calculus with rewrite rules to model the inverse of the evaluation functional. Among its operational properties figures a standardization theorem that formally establishes adequacy of implementation in functional programming languages. An example implementation in `Haskell` is provided. The relation to usual type-directed Normalization by Evaluation is highlighted, using a short analysis of η-expansion that leads to a perspicuous strong normalization and confluence proof for $\beta\eta^{\uparrow}$-reduction as a byproduct.

Introduction

Normalization by Evaluation uses the evaluation mechanism of a metalanguage to normalize terms, typically of the λ-calculus. By means of an interpretation function $[\![\cdot]\!]_\xi$, terms are embedded into this metalanguage; an "inverse of the evaluation functional" (Berger and Schwichtenberg, 1991) $\downarrow$ serves to recover terms from the semantics. The two essential properties of these functions are

— *Soundness:* $r \to s \;\Rightarrow\; [\![r]\!]_\xi = [\![s]\!]_\xi$, and
— *Reproduction:* r in normal form $\Rightarrow\; \downarrow[\![r]\!]_\uparrow = r$ with $\uparrow$ a special valuation.

These two properties ensure that $\downarrow[\![r]\!]_\uparrow$ actually yields a normal form of r if it exists.

The aim of this article is to extend the mechanism of Normalization by Evaluation to the untyped λ-calculus, where normal forms need not exist.[1]

As a starting point, recall the naive set-theoretic model of Normalization by Evaluation for the prototypical case of the simply typed λ-calculus: base types ι are interpreted by the set $[\![\iota]\!] := \mathrm{NF}_\iota$ of β-normal λ-terms of type ι, while arrow types $\rho \to \sigma$ are inductively interpreted as (an appropriate subset of) the set-theoretic function space $[\![\rho]\!] \to [\![\sigma]\!]$.

[†] Supported by the Graduiertenkolleg "Logik in der Informatik" of the Deutsche Forschungsgemeinschaft.

[1] In particular, we cannot "extract" (Berger, 1993) the normalization function from a proof of normalization.

Relative to a valuation ξ of variables, the semantics $[\![r]\!]_\xi$ of a term r is defined as usual by[2]

$$[\![x]\!]_\xi := \xi x, \qquad [\![rs]\!]_\xi := [\![r]\!]_\xi [\![s]\!]_\xi, \qquad [\![\lambda x r]\!]_\xi := \lambda\!\!\lambda X [\![r]\!]_{\xi, x:=X}.$$

This model is sound w.r.t. β-reduction in the sense defined above. The functions

$$\uparrow_\rho : \{x\vec{r} \mid \vec{r} \text{ normal}\} \to [\![\rho]\!] \qquad \text{and} \qquad \downarrow_\rho : [\![\rho]\!] \to \mathrm{NF}_\rho$$

are defined by mutual recursion on ρ:

$$(*) \qquad \begin{array}{lclclcll} \uparrow_\iota r & := & r, & \uparrow_{\rho\to\sigma} r & := & \lambda\!\!\lambda X \uparrow_\sigma (r \downarrow_\rho X), & \\ \downarrow_\iota r & := & r, & \downarrow_{\rho\to\sigma} R & := & \lambda x \downarrow_\sigma (R \uparrow_\rho x), & x \text{ new.}^3 \end{array}$$

Simple properties of these functions are

$$\begin{array}{rcl} \downarrow_{\rho\to\sigma} \lambda\!\!\lambda X R & = & \lambda x \downarrow_\sigma ((\lambda\!\!\lambda X R) \uparrow_\rho x) \\ & = & \lambda x \downarrow_\sigma (R[X := \uparrow_\rho x]), \\ \downarrow_\iota \uparrow_\iota r & = & r, \\ (\uparrow_{\rho\to\sigma} r) S & = & (\lambda\!\!\lambda X \uparrow_\sigma (r \downarrow_\rho X)) S \\ & = & \uparrow_\sigma (r \downarrow_\rho S). \end{array}$$

The crucial observation is that these equations define the behaviour of the application function and $\downarrow$ just as well as $(*)$ did, even when the type annotations are not available — the only information needed is whether their argument is a function (and therefore by extensionality of the form $\lambda\!\!\lambda X R$) or a lifted syntax object $\uparrow r$:

— The *reification* of a function $\lambda\!\!\lambda X R$ is $\lambda x \downarrow R[X := \uparrow x]$ with a new x.[4]

— The *reification* of $\uparrow r$ yields just r.

— *Application* of $\uparrow r$ to an object S results in $\uparrow (r \downarrow S)$.

These rules allow a type-free reformulation of Normalization by Evaluation which then can serve to normalize any untyped λ-term, hence in particular terms of system F and stronger type systems, thus strengthening previous results of extensions of Normalization by Evaluation to stronger type systems (Altenkirch et al., 1996).

Note that Mogensen (1999) also studied an untyped algorithm for Normalization by Evaluation, building on Goldberg's work (1996,2000) on Gödelisation in the λ-calculus. Since his definition is more involved, a study of the operational behavior was not attempted and confluence could only be conjectured.

Two-level λ-calculus. Non-syntactic models of Normalization by Evaluation, as the one sketched above, need to provide a mechanism to determine the free variables of their semantic objects in order to justify the definition of $\downarrow$ on functions. A solution for a set-theoretic semantics has been presented by Berger and Schwichtenberg (1991). Berger

[2] We use the notation $\lambda\!\!\lambda X R$ for set-theoretic functional abstraction, employing R, S, T for objects of the set-theoretic metalanguage.

[3] Much effort has been invested into finding such a new x, the reason being that a function in $[\![\rho]\!] \to [\![\sigma]\!]$ is not equipped with a natural notion of free variables. In our syntactical model this is not a problem, so we ignore the issue in this introduction.

[4] With the same shortcomings as above.

et al. (1998) use term-families in a domain-theoretic setting, while other publications (Altenkirch et al., 1995; Coquand and Dybjer, 1997; Čubrić et al., 1998) give solutions for a category-theoretic approach. Although all mentioned methods would apply to the algorithm sketched above, semantic issues tend to blur the operational aspects of Normalization by Evaluation. For the operational analysis[5] we thus formulate the algorithm in the purely syntactic framework of a two-level λ-calculus. It differs from other published two-level models (Danvy, 1998; Vestergaard, 2001) by its explicit representation of the reification/reflection functions: the lower level Λ_1 consists of the object-level λ-calculus Λ and reifications $\downarrow R$ of meta-level terms $R \in \Lambda_2$, while Λ_2 has a constructor $\uparrow r$ that embeds $r \in \Lambda_1$. The normalization algorithm is formulated by rewrite rules, so that we can analyze and derive operational properties such as confluence and standardization, using methods from higher-order term rewriting theory. As a corollary to the standardization theorem the implementation of the algorithm in usual functional programming languages is provably correct and can be shown to appropriately deal with non-terminating computations (Böhm trees) in lazy languages.

Adding types. It is shown that the addition of types leads to strong normalization of the rewrite rules and the computation of long $\beta\eta$-normal forms. In order to relate our approach to customary type-directed expositions, it is explained how the rewrite rules are emulated by type-directed Normalization by Evaluation. This is made possible through a perspicuous characterization of η-expansive normal forms that also permits — as a byproduct — a comparably short proof of confluence and strong normalization of combined $\beta\eta\Uparrow$-reduction in the simply typed λ-calculus.

Implementation in `Haskell`. We demonstrate the translation of the algorithm as given in the rewrite model into a program by an example implementation in the lazy functional programming language `Haskell`, where the coinductive notion of Böhm trees requires no further programming effort. In order to avoid the perils of α-equality, the object-level λ-calculus is implemented in a de Bruijn-language (Bruijn, 1972).

Outline of the contents. Section 1 presents the two-level calculus $\Lambda_{1,2}$ together with its rewriting rules. Confluence follows from orthogonality by results of higher-order term rewriting theory. Finally, an inductive term and normal form characterization is provided that serves as a starting point for the standardization results of section 3. In section 2 it is shown that the evaluation function $[\![\cdot]\!]_\xi : \Lambda \to \Lambda_2$ leads to simulation (which instantiates to soundness in the sense sketched above) and reproduction. Section 3 uses a technique of Joachimski and Matthes (2000) to devise a notion of standard reduction sequence that is shown to be equivalent to reduction ("standardization"), so that the above mentioned corollaries on implementation and Böhm trees can be drawn. Section 4 establishes a strong form of correctness for computations in the rewrite model of Normalization by Evaluation, which applies even to diverging computations. Furthermore a normal form property of the model is shown. Section 5 is an excursion into the reduction behavior of

[5] Note that important operational aspects like adequacy of the implementation in a functional programming language have not been formally addressed in the literature before.

η-expansion $\to_{\eta\Uparrow}$, showing strong normalization and confluence of $\to_{\beta\eta\Uparrow}$. In section 6 we give a typed version of our normalization algorithm and compare it to usual type-directed Normalization by Evaluation. Section 7 discusses an implementation in `Haskell`.

Acknowledgment. We are most grateful to Ralph Matthes for his valuable remarks on this work.

1. The Calculus

We define a two-level λ-calculus with explicit representations of the reification function $\downarrow$. In its lower level the calculus contains the usual λ-calculus.

1.1. *Notational conventions*

We use concatenation $\rightarrowtail\rightarrowtail'$ of binary relation symbols to denote their relational composition $\rightarrowtail' \circ \rightarrowtail$ and write $\rightarrowtail^+$ ($\rightarrowtail^*$) for the transitive (reflexive transitive) closure of $\rightarrowtail$. $\rightarrowtail^n$ denotes the n-fold composition of $\rightarrowtail$ with itself. Lists of the form $e_1, \ldots, e_n$ are written $\vec{e}$, including the case $\varepsilon := \vec{e}$ if $n = 0$. We write $e, \vec{e}$ to prefix e to the list $\vec{e}$ and identify the one-element list e, ε with e.

1.2. *Terms*

We presuppose an infinite supply of object variables $x, y, z \in o\mathcal{V}$ and distinct meta-variables $X, Y, Z \in m\mathcal{V}$. Terms of the usual λ-calculus Λ are given by the grammar

$$\Lambda \ni \underline{r}, \underline{s}, \underline{t} ::= x \mid \underline{r}\,\underline{s} \mid \lambda x \underline{r}.$$

The two-level λ-calculus $\Lambda_{1,2} := \Lambda_1 \cup \Lambda_2$ consists of two simultaneously defined term sets

$$\begin{array}{llll} \Lambda_1 & \ni & r, s, t & ::= & x \mid rs \mid \lambda x r \mid R\Downarrow, \\ \Lambda_2 & \ni & R, S, T & ::= & X \mid RS \mid \lambda X R \mid \uparrow r. \end{array}$$

Since Λ is a sub-grammar of Λ_1 we identify it with the respective subset, so $\Lambda \subseteq \Lambda_1$.[6] As usual, we let application in all calculi associate to the left, writing $r\vec{r}$ for the iterated elimination of r with the elements of the list $\vec{r}$ (and similarly for $R\vec{R}$). Repeated introductions $\lambda x_1 \ldots \lambda x_n r$ are written $\lambda \vec{x} r$.

Note that we have used $\Downarrow$ in postfix notation instead of $\downarrow$, since we consider $\downarrow$ an elimination rather than an introduction (see section 3 for details). In order to recover the usual notation $\downarrow R$ we define $\downarrow\vec{R}$ for a list $\vec{R}$ recursively by $\downarrow\varepsilon := \varepsilon$, $\downarrow(R, \vec{R}) := (R\Downarrow, \downarrow\vec{R})$, in particular $\downarrow R = R\Downarrow$. The $\uparrow$-constructor is extended to lists in the canonical way.

The λ-constructors bind their variable argument. We identify terms that differ only in names of bound variables, adopting the standard variable conventions. Substitution is denoted by $r[x := s], R[X := S]$. $f\mathcal{V}R$, $f\mathcal{V}r$ stand for the free variables of R, r, respectively. A term is called *meta-closed*, if its free variables are in $o\mathcal{V}$.

[6] And underlined $\underline{r}, \underline{s}, \underline{t}$ are elements in $\Lambda \cap \Lambda_1$.

1.3. *Reduction*

The symbol $\to_{\underline{\beta}}$ is used for β-reduction in Λ, generated from the elementary reduction rule $(\lambda x\underline{r})\underline{s} \mapsto_{\underline{\beta}} \underline{r}[x := \underline{s}]$ by means of the compatible term closure in Λ.

In $\Lambda_{1,2}$ we use rewrite rules for $\downarrow$ and $\uparrow$ and β-reduction in order to simulate the $\underline{\beta}$-reduction in Λ (see the introduction for a motivation of these rules). *Elementary reduction rules* are

$$\begin{array}{rcll} (\lambda XR)S & \mapsto_\beta & R[X := S], & \\ (\uparrow r)S & \mapsto_a & \uparrow(r\downarrow S), & \\ \downarrow\lambda XR & \mapsto_d & \lambda x\downarrow(R[X := \uparrow x]), & x \text{ new}, \\ \downarrow\uparrow r & \mapsto_d & r. & \end{array}$$

The *term closure* $\to_\phi$ of the elementary reduction $\mapsto_\phi$ ($\phi \in \{\beta, d, a\}$) is obtained by

$$\frac{r \to_\phi r'}{\lambda x r \to_\phi \lambda x r'} \qquad \frac{r \to_\phi r'}{rs \to_\phi r's} \qquad \frac{s \to_\phi s'}{rs \to_\phi rs'} \qquad \frac{R \to_\phi R'}{R\Downarrow \to_\phi R'\Downarrow}$$

$$\frac{R \to_\phi R'}{\lambda X R \to_\phi \lambda X R'} \qquad \frac{R \to_\phi R'}{RS \to_\phi R'S} \qquad \frac{S \to_\phi S'}{RS \to_\phi RS'} \qquad \frac{r \to_\phi r'}{\uparrow r \to_\phi \uparrow r'}$$

Define $\to_s := \to_a \cup \to_d$ *(syntactic reduction)* and $\to := \to_\beta \cup \to_s$. NF_ϕ denotes the set of normal terms w.r.t. $\to_\phi$ for $\phi \in \{\underline{\beta}, \beta, a, d, s\}$; NF is the set of normal terms w.r.t. $\to$, i.e., $\mathrm{NF} := \mathrm{NF}_\beta \cap \mathrm{NF}_s$. $=_\phi$ denotes the reflexive transitive and symmetric closure of $\to_\phi$.

1.4. *Inductive characterizations*

In Λ, terms and normal forms are inductively characterized by the grammars

$$\begin{array}{rcrcl} \Lambda & \ni & \underline{r} & ::= & x\vec{\underline{r}} \mid (\lambda x\underline{r})\vec{\underline{r}}, \\ \mathrm{NF}_{\underline{\beta}} & \ni & \underline{r} & ::= & x\vec{\underline{r}} \mid \ \lambda x\underline{r}. \end{array}$$

In the same spirit, a two-level characterization can be shown for $\Lambda_{1,2}$.

Proposition 1.4.1. Terms and normal forms are inductively characterized by

$$\begin{array}{rcrcl} \Lambda_1 & \ni & r & ::= & x\vec{r} \mid (\lambda x r)\vec{r} \mid R\Downarrow\vec{r}, \\ \Lambda_2 & \ni & R & ::= & X\vec{R} \mid (\uparrow r)\vec{R} \mid (\lambda XR)\vec{R}, \\ \mathrm{NF} \cap \Lambda_1 & \ni & r & ::= & x\vec{r} \mid (\lambda x r)\vec{r} \mid (X\vec{R})\Downarrow\vec{r}, \\ \mathrm{NF} \cap \Lambda_2 & \ni & R & ::= & X\vec{R} \mid \uparrow r \mid \lambda XR. \end{array}$$

Remark that a meta-closed term $r \in \mathrm{NF}$ is already in Λ.

1.5. *Confluence*

The calculus is a higher order, left-linear pattern rewrite system (Oostrom, 1997). Standard results in higher order rewriting theory (see loc.cit.) allow to infer confluence from orthogonality, which requires the easy verification that there are no critical pairs.[7]

2. Normalization by Evaluation

We proceed along the lines sketched in the introduction, defining evaluation first and then deriving soundness from the simulation of $\underline{\beta}$-reductions on the interpretation of Λ-terms in Λ_2.

2.1. *Evaluation*

A *valuation* ξ is a mapping $\xi : o\mathcal{V} \to \Lambda_2$ that differs from $\uparrow$ at only finitely many object variables x. The extension of a valuation is defined by

$$(\xi, x := R)y := \begin{cases} R & \text{if } x = y, \\ y & \text{otherwise.} \end{cases}$$

We set $\mathit{fV}_\xi \underline{r} := \bigcup_{x \in \mathit{fV}\underline{r}} \mathit{fV}(\xi x)$. Relative to a valuation ξ, the *evaluation* function $[\![-]\!]_\xi : \Lambda \to \Lambda_2$ is defined recursively by

$$\begin{aligned} [\![x]\!]_\xi &:= \xi x, \\ [\![\underline{rs}]\!]_\xi &:= [\![\underline{r}]\!]_\xi \, [\![\underline{s}]\!]_\xi, \\ [\![\lambda x \underline{r}]\!]_\xi &:= \lambda X [\![\underline{r}]\!]_{\xi, x:=X}, \qquad X \notin \mathit{fV}_\xi \underline{r}. \end{aligned}$$

Remark that $[\![\underline{r}]\!]_\xi$ contains only the variables $\mathit{fV}_\xi \underline{r}$. Thus $[\![\underline{r}]\!]_\uparrow$ is meta-closed.

Proposition 2.1.1. $[\![\underline{r}[x := \underline{s}]]\!]_\xi = [\![\underline{r}]\!]_{\xi, x:=[\![\underline{s}]\!]_\xi}$.

Proof. Induction on $\underline{r}$. □

Lemma 2.1.2 (Simulation). $\underline{r} \to_{\underline{\beta}} \underline{s} \implies [\![\underline{r}]\!]_\xi \to_\beta [\![\underline{s}]\!]_\xi$.

Proof. Induction on $\to_{\underline{\beta}}$. We treat only the case of an elementary reduction:

$$\begin{aligned} [\![(\lambda x \underline{r})\underline{s}]\!]_\xi &= (\lambda X [\![\underline{r}]\!]_{\xi, x:=X}) [\![\underline{s}]\!]_\xi \\ &\mapsto_\beta [\![\underline{r}]\!]_{\xi, x:=X}[X := [\![\underline{s}]\!]_\xi] \\ &= [\![\underline{r}]\!]_{\xi, x:=[\![\underline{s}]\!]_\xi} && X \notin \mathit{fV}_\xi \underline{r}, \\ &= [\![\underline{r}[x := \underline{s}]]\!]_\xi && \text{by proposition 2.1.1.} \end{aligned}$$

□

[7] Of course, a proof using developments can also be reconstructed by hand.

2.2. *Reproduction*

Example 2.2.1. Using the abbreviation $[\![\underline{r}]\!] := [\![\underline{r}]\!]_{\uparrow}$ we calculate

$$\downarrow[\![\lambda x\underline{r}]\!] = \downarrow\lambda X[\![\underline{r}]\!]_{\uparrow,x:=X} \to_d \lambda x\downarrow[\![\underline{r}]\!]_{\uparrow,x:=X}[X := \uparrow x] = \lambda x\downarrow[\![\underline{r}]\!].$$

Proposition 2.2.2. $(\uparrow r)\vec{S} \to_a^* \uparrow(r\downarrow\vec{S})$.

Proof. Induction on $\vec{S}$, using $(\uparrow r)S\vec{S} \to_a \uparrow(r\downarrow S)\vec{S}$. □

Lemma 2.2.3. $\underline{r} \in \mathrm{NF}_{\underline{\beta}} \implies \downarrow[\![\underline{r}]\!] \to_s^* \underline{r}$.

Proof. Induction on $\mathrm{NF}_{\underline{\beta}}$.

$$\begin{array}{rcll}
\downarrow[\![x\vec{\underline{r}}]\!] & = & \downarrow((\uparrow x)[\![\vec{\underline{r}}]\!]) & \\
& \to_a^* & \downarrow\uparrow(x\downarrow[\![\vec{\underline{r}}]\!]) & \text{by proposition 2.2.2,} \\
& \to_d & x\downarrow[\![\vec{\underline{r}}]\!] & \\
& \to_s^* & x\vec{\underline{r}} & \text{by IH.} \\
\downarrow[\![\lambda x\underline{r}]\!] & \to_d & \lambda x\downarrow[\![\underline{r}]\!] & \text{see example 2.2.1,} \\
& \to_s^* & \lambda x\underline{r} & \text{by IH.}
\end{array}$$

□

Using simulation and lemma 2.2.3 we obtain

Corollary 2.2.4. $\underline{r} \to_{\underline{\beta}}^* \underline{s} \in \mathrm{NF}_{\underline{\beta}} \implies \downarrow[\![\underline{r}]\!] \to^* \underline{s}$.

3. Standardization

The aim of this section is to establish that in the rewrite system of section 1, the reduction strategies of various functional programming languages are sufficient to achieve Normalization by Evaluation as described in the last section. In particular, it is shown that reductions need not be performed under meta-abstractions and can follow a call-by-name strategy. Furthermore, a demand-driven evaluation strategy is proved sufficient to compute the defined parts of Böhm trees.

These results are derived from a general standardization theorem which states that all reduction sequences in the calculus $\Lambda_{1,2}$ can be rearranged into a so-called standard order.

The section is divided into three parts:

— Following ideas of Joachimski and Matthes (2000) we first introduce an inductive notion $\rightsquigarrow$ of standard reduction sequences. The standardization theorem shows that $\rightsquigarrow$ is extensionally equal to $\to^*$. However, the intensional structure of $\rightsquigarrow$ provides a powerful notion of induction that is used in the following arguments.

— In the second step we define a restriction $\twoheadrightarrow$ of $\rightsquigarrow$ which captures exactly the reduction behaviour of functional programming languages. That $\twoheadrightarrow$ always leads to normal forms and thus is adequate for Normalization by Evaluation is a consequence of the standardization theorem.

— Demand-driven evaluation mechanisms allow that certain parts of a term are not further reduced, unless evaluation is necessary for the computation of the output. This is captured by a variant $\twoheadrightarrow$ of $\twoheadrightarrow\!\!\!\!\!\rightarrow$, which can be iteratively applied to compute the defined parts of the Böhm tree, thus modeling the behaviour of lazy reduction.

3.1. *Generalized eliminations*

The constructor $\uparrow$ of $\Lambda_{1,2}$ can be interpreted as the *introduction* of Λ_1-terms into Λ_2. In this natural deduction reading, $\Downarrow$ is considered an *elimination* and thus written postfix. In order to capture this intuition and adapt the standardization argument of the λ-calculus to $\Lambda_{1,2}$ we introduce a generalization of eliminations: let E, F stand for elements of $\Lambda_{1,2} \cup \{\Downarrow\}$. We write RE and rE for the corresponding term of $\Lambda_{1,2}$ if and only if the expression is defined. For instance, $R{\Downarrow}s$ might be denoted by REF with $E := \Downarrow$ and $F := s$, and if we write rE, then this presupposes that $E \in \Lambda_1$.

3.2. *Standard reduction*

Using $\Downarrow \rightsquigarrow \Downarrow$ for truth and $\vec{E} \rightsquigarrow \vec{E'}$ as an abbreviation for $E_1 \rightsquigarrow E_1'$ & $\ldots$ & $E_n \rightsquigarrow E_n'$, we define inductively

$$\frac{\vec{r} \rightsquigarrow \vec{r'}}{x\vec{r} \rightsquigarrow x\vec{r'}}\,(v) \qquad \frac{\vec{E} \rightsquigarrow \vec{E'}}{X\vec{E} \rightsquigarrow X\vec{E'}}\,(V)$$

$$\frac{r, \vec{s} \rightsquigarrow r', \vec{s'}}{(\lambda x r)\vec{s} \rightsquigarrow (\lambda x r')\vec{s'}}\,(\lambda_1) \qquad \frac{R, \vec{E} \rightsquigarrow R', \vec{E'}}{(\lambda X R)\vec{E} \rightsquigarrow (\lambda X R')\vec{E'}}\,(\lambda_2)$$

$$\frac{(\lambda x {\downarrow} R[X := {\uparrow} x])\vec{r} \rightsquigarrow t}{(\lambda X R){\Downarrow}\vec{r} \rightsquigarrow t}\,(d_1) \qquad \frac{R[X := S]\vec{E} \rightsquigarrow E}{(\lambda X R)S\vec{E} \rightsquigarrow E}\,(\beta)$$

$$\frac{r\vec{s} \rightsquigarrow t}{({\uparrow} r){\Downarrow}\vec{s} \rightsquigarrow t}\,(d_2) \qquad \frac{{\uparrow}(r{\downarrow}S)\vec{E} \rightsquigarrow E}{({\uparrow} r)S\vec{E} \rightsquigarrow E}\,(a)$$

$$\frac{r, \vec{E} \rightsquigarrow r', \vec{E'}}{({\uparrow} r)\vec{E} \rightsquigarrow ({\uparrow} r')\vec{E'}}\,(\uparrow)$$

A derivation of $r \rightsquigarrow s$ allows to devise various reduction sequences from r to s. Their common characteristic is that head redexes — as identified by the inductive characterization of terms — are either executed first (by (β),(a),(d_1),(d_2)) or never touched afterwards, performing only reductions in the components of the redex. The deconstruction rules (v),(V),(λ_1),(λ_2),$(\uparrow)$ do not stipulate a particular order of reductions. For instance, in $xrs \rightsquigarrow xr's'$ we can choose to either first execute all reductions in r before those in s, or vice versa, or even mix the reductions at random.

It should be noted that $\rightsquigarrow$ is closed under abstraction as a degenerated case of rules (λ_1) and (λ_2).

Proposition 3.2.1. $\rightsquigarrow \subseteq \rightarrow^*$.

Proof. Induction on $\rightsquigarrow$ using, say, left-to-right reduction order for the cases in which there is a choice: $(v), (V), (\lambda_1), (\lambda_2), (\uparrow)$. □

Lemma 3.2.2.
(1). $R, E \rightsquigarrow R', E' \quad \Longrightarrow \quad RE \rightsquigarrow R'E'$,
$\qquad r, s \rightsquigarrow r', s' \quad \Longrightarrow \quad rs \rightsquigarrow r's'$,
(2). $r \rightsquigarrow r$ and $R \rightsquigarrow R$.

Proof. (1). Easy induction on $\rightsquigarrow$. (2). Simultaneous induction on r and R, using proposition 1.4.1 of subsection 1.4 and (1). □

3.3. *Completeness*

The previous subsection established soundness of $\rightsquigarrow$ w.r.t. $\to^*$. Standardization amounts to completeness: every reduction sequence can be turned into a standard one.

Lemma 3.3.1. $R, S \rightsquigarrow R', S' \implies R[X := S] \rightsquigarrow R'[X := S']$.

Proof. Induction on $R \rightsquigarrow R'$. We demonstrate the only non-trivial case where $X\vec{E} \rightsquigarrow X\vec{E'}$ has been concluded from $\vec{E} \rightsquigarrow \vec{E'}$: the induction hypothesis furnishes $\vec{E}[X := S] \rightsquigarrow \vec{E'}[X := S']$. Applying (1) repeatedly, we obtain $S\vec{E}[X := S] \rightsquigarrow S'\vec{E'}[X := S']$. □

Theorem 3.3.2. $\rightsquigarrow\to \;\subseteq\; \rightsquigarrow$.

Proof. Induction on $\rightsquigarrow$.
Case $(\downarrow\lambda XR)\vec{r} \rightsquigarrow (\downarrow\lambda XR')\vec{r'} \to_d (\lambda x{\downarrow}R'[X := \uparrow x])\vec{r'}$ from $R, \vec{r} \rightsquigarrow R', \vec{r'}$.

$$
\begin{array}{rcll}
\uparrow x & \rightsquigarrow & \uparrow x & \text{by (2)}, \\
R[X := \uparrow x] & \rightsquigarrow & R'[X := \uparrow x] & \text{by lemma 3.3.1}, \\
\downarrow R[X := \uparrow x] & \rightsquigarrow & \downarrow R'[X := \uparrow x] & \text{by (1)}, \\
(\lambda x{\downarrow}R[X := \uparrow x])\vec{r} & \rightsquigarrow & (\lambda x{\downarrow}R'[X := \uparrow x])\vec{r'} & \text{by } (\lambda_1), \\
(\downarrow\lambda XR)\vec{r} & \rightsquigarrow & (\lambda x{\downarrow}R'[X := \uparrow x])\vec{r'} & \text{by } (d_1).
\end{array}
$$

Case $(\lambda XR)S\vec{E} \rightsquigarrow (\lambda XR')S'\vec{E'} \to_\beta R'[X := S']\vec{E'}$ from

$$
\begin{array}{rcll}
R, S, \vec{E} & \rightsquigarrow & R', S', \vec{E'}. & \\
R[X := S] & \rightsquigarrow & R'[X := S'] & \text{by lemma 3.3.1}, \\
R[X := S]\vec{E} & \rightsquigarrow & R'[X := S']\vec{E'} & \text{by (1)}, \\
(\lambda XR)S\vec{E} & \rightsquigarrow & R'[X := S']\vec{E'} & \text{by } (\beta).
\end{array}
$$

Case $(\uparrow r)\Downarrow\vec{r} \rightsquigarrow (\uparrow r')\Downarrow\vec{r'} \to_d r'\vec{r'}$: use (1) and (d_2).
Case $(\uparrow r)S\vec{E} \rightsquigarrow (\uparrow r')S'\vec{E'} \to_a \uparrow(r'{\downarrow}S')\vec{E'}$: invoke (1) repeatedly and use rule (a).
The cases $(\beta), (d_1), (d_2), (v), (V), (a), (\lambda_1)$ and the remaining cases for (λ_2) and $(\uparrow)$ are simple applications of the induction hypothesis. □

Corollary 3.3.3 (Standardization). $\rightsquigarrow \;=\; \to^*$.

Proof. $\subseteq$ was the subject of proposition 3.2.1 For $\supseteq$ we proceed by induction on the length of a given reduction sequence, employing (2) for the case of an empty sequence and the theorem for the step case. □

Standardization provides a new induction principle for $\to^*$. This can be used to prove adequacy of standard reduction strategies, in our case, of the reduction behaviour of functional programming languages.

3.4. *Restricted standard reduction*

The relation $\twoheadrightarrow$ is generated by the rules of $(\beta), (d_1), (d_2), (a), (v)$ of $\leadsto$ and the rule

$$\frac{r \twoheadrightarrow r'}{\lambda x r \twoheadrightarrow \lambda x r'}\,(\lambda)$$

By extracting $r \to^* s$ from a derivation of $r \twoheadrightarrow s$ we arrive at reduction sequences that never reduce under meta-abstractions. The order of evaluations is still not fixed, but can be chosen, e.g., left-to-right. Thus $\twoheadrightarrow$ can be used to model the evaluation order of functional programming languages.

So the goal is to show that $\twoheadrightarrow$ actually suffices to compute the normal form, if it exists.

Proposition 3.4.1. $\Lambda_1 \times \mathrm{NF}_{\underline{\beta}} \supseteq \twoheadrightarrow \subseteq \leadsto$.

Proof. Easy induction on $\twoheadrightarrow$. □

Lemma 3.4.2. $r \leadsto \underline{r} \in \mathrm{NF}_{\underline{\beta}} \implies r \twoheadrightarrow \underline{r}$.

Proof. Inductive verification that the rules $(V), (\uparrow), (\lambda_2)$ cannot have been applied and (λ_1) is only used in the form of (λ). □

Corollary 3.4.3. $\underline{r} \to^*_{\underline{\beta}} \underline{s} \in \mathrm{NF}_{\underline{\beta}} \implies \downarrow[\![\underline{r}]\!] \twoheadrightarrow \underline{s}$.

3.5. *Böhm trees*

A closer inspection reveals that lazy evaluation is also sufficient to successively develop non-normalizing terms. The coinductive notion of Böhm trees (Barendregt, 1977) is usually invoked to model iterative head-normalization: the Böhm tree Ξ of r is either undefined (if r has no head normal form) or $\lambda\vec{x}(x\vec{\Xi})$ if $\vec{\Xi}$ are the Böhm trees of $\vec{r}$ where $\lambda\vec{x}(x\vec{r})$ is a head normal form of r. We will show below that (the defined parts of) Böhm trees can be computed by a lazy reduction strategy that never reduces under meta-abstractions.

Proposition 3.5.1. $\underline{r} \to^*_{\underline{\beta}} \lambda\vec{x}(x\underline{\vec{r}}) \implies \downarrow[\![\underline{r}]\!] \to^* \lambda\vec{x}(x\downarrow[\![\underline{\vec{r}}]\!])$.

Proof. Lemma 2.1.2 shows $[\![\underline{r}]\!] \to^*_\beta [\![\lambda\vec{x}(x\underline{\vec{r}})]\!]$, so

$$\begin{array}{rcll}
\downarrow[\![\underline{r}]\!] & \to^*_\beta & \downarrow[\![\lambda\vec{x}(x\underline{\vec{r}})]\!] & \\
 & \to^*_d & \lambda\vec{x}\downarrow[\![x\underline{\vec{r}}]\!] & \text{by example 2.2.1,} \\
 & = & \lambda\vec{x}\downarrow((\uparrow x)[\![\underline{\vec{r}}]\!]) & \\
 & \to^*_a & \lambda\vec{x}\downarrow\uparrow(x\downarrow[\![\underline{\vec{r}}]\!]) & \text{by proposition 2.2.2,} \\
 & \to_d & \lambda\vec{x}(x\downarrow[\![\underline{\vec{r}}]\!]). &
\end{array}$$

□

Definition. $\twoheadrightarrow$ is generated by the rules $(\beta), (d_1), (d_2), (a), (\lambda)$ of $\twoheadrightarrow\!\!\!\!\rightarrow$ and *reflexivity* on variable eliminations $x\vec{r} \twoheadrightarrow x\vec{r}$.

$\twoheadrightarrow$ captures a call-by-name strategy that traverses eventual object level abstractions, but no applications. Iterative application of $\twoheadrightarrow$ to appropriate subterms models the evaluation mechanism of a call-by-need functional programming language: If a computation reaches a term $x\vec{r}$, the external need (for instance the user's request for display of a subterm) determines which of the r_i is going to be further evaluated by $\twoheadrightarrow$-reductions.

Lemma 3.5.2. $r \rightsquigarrow \lambda\vec{x}(x\vec{r}) \implies \exists\vec{s}.r \twoheadrightarrow \lambda\vec{x}.x\vec{s} \rightarrow^* \lambda\vec{x}.x\vec{r}$.

Proof. Induction on $\rightsquigarrow$. The cases $(\lambda_2), (V), (\uparrow)$ cannot have been used due to the form of the reduct. The reduction cases $(\beta), (a), (d_1), (d_2)$ are treated by the induction hypothesis and the corresponding rules of $\twoheadrightarrow$.

Case (λ_1). The trailing vector of eliminations has to be empty due to the form of the reduct, so we can apply (λ) to the induction hypothesis.

Case (v): $x\vec{s} \rightsquigarrow x\vec{r}$. Use reflexivity $x\vec{s} \twoheadrightarrow x\vec{s}$ and soundness of $\rightsquigarrow$. □

Corollary 3.5.3. $\underline{r} \rightarrow^*_{\underline{\beta}} \underline{s} \in \mathrm{NF}_{\underline{\beta}} \implies \downarrow[\![\underline{r}]\!] \twoheadrightarrow \underline{s}$.

Proof. Induction on $\underline{s}$. □

Theorem 3.5.4.

$$\begin{array}{lll} & \underline{r} =_{\underline{\beta}} \lambda\vec{x}(x\vec{\underline{r}}) & \& \;\; s \rightarrow^* \downarrow[\![\underline{r}]\!] \\ \implies \exists\vec{\underline{t}}, \vec{s}.s \twoheadrightarrow \lambda\vec{x}(x\vec{s}) \;\&\; \vec{\underline{r}} =_{\underline{\beta}} \vec{\underline{t}} & & \& \;\; \vec{s} \rightarrow^* \downarrow[\![\vec{\underline{t}}]\!]. \end{array}$$

Proof.

$$\begin{array}{ll} \underline{r} \rightarrow^*_{\underline{\beta}} \lambda\vec{x}(x\vec{\underline{t}}) \;{}^*_{\underline{\beta}}\!\!\leftarrow \lambda\vec{x}(x\vec{\underline{r}}) & \text{by the Church-Rosser prop. of } \rightarrow_{\underline{\beta}}, \\ s \rightarrow^* \downarrow[\![\underline{r}]\!] \rightarrow^* \lambda\vec{x}(x\downarrow[\![\vec{\underline{t}}]\!]) & \text{by proposition 3.5.1,} \\ s \rightsquigarrow \lambda\vec{x}(x\downarrow[\![\vec{\underline{t}}]\!]) & \text{by standardization,} \\ s \twoheadrightarrow \lambda\vec{x}(x\vec{s}) \rightarrow^* \lambda\vec{x}(x\downarrow[\![\vec{\underline{t}}]\!]) & \text{by lemma 3.5.2.} \end{array}$$

□

This theorem can be used to calculate the defined parts of the Böhm tree of $\underline{r}$ as follows. Let $\lambda\vec{x}(x\vec{\underline{r}})$ be a head normal form of $\underline{r}$. Starting with $s := \downarrow[\![\underline{r}]\!]$, the theorem yields $s \twoheadrightarrow \lambda\vec{x}(x\vec{s})$ with $\vec{s} \rightarrow^* \downarrow[\![\vec{\underline{t}}]\!]$ and $\vec{\underline{t}} =_{\underline{\beta}} \vec{\underline{r}}$. In other words, call-by-name reduction leads to the correct beginning of the Böhm tree and $\vec{s}$ are such that they can again be fed into the theorem to compute the defined subtrees.

4. Correctness of Results

By virtue of confluence and corollary 2.2.4 any normalizing term reduces to the normal form computed for it by the Normalization by Evaluation algorithm. Theorem 3.5.4 generalizes this to the case where at least a head normal form exists. In this section we prove that the computation in the rewrite model diverges, if the term to be normalized does not have a head normal form. In a constructive reading this amounts to

$$\downarrow[\![\underline{t}]\!] \rightarrow^* \lambda\vec{x}(x\vec{R}) \implies \underline{t} \rightarrow^* \lambda\vec{x}(x\vec{\underline{t}}),$$

where $\underline{\vec{t}}$ are the terms represented by the semantical objects $\vec{R}$ (the *kernel* of $\vec{R}$).

4.1. *Kernel*

Semantical objects, i.e., terms of Λ_2, model states in a computation of object terms in Λ_1. If one strips a term in Λ_2 off the administrative overhead imposed by the constructors $\uparrow$ and $\downarrow$, one arrives at its *kernel*. In other words, the kernel is the term a semantical object denotes.

Let $|r|^{\uparrow}$ and $|R|^{\uparrow}$ be the height of r and R, respectively, not counting the constructor $\uparrow$. By recursion on $|r|^{\uparrow}$ (with side recursion on $|R|^{\uparrow}$ for the case $R\Downarrow$), we define the *kernel* $r^* \in \Lambda$ only for meta-closed r as follows.

$$x^* := x, \quad (rs)^* := r^* s^*, \quad (R\Downarrow)^* := R^*, \quad (\lambda x r)^* := \lambda x r^*,$$
$$(RS)^* := R^* S^*, \quad \uparrow r^* := r^*, \quad (\lambda X R)^* := \lambda x (R[X := \uparrow x])^*,$$

where in the last clause we require the x to be new.

Proposition 4.1.1. $r \to_s s \implies r^* = s^*$.

Proof. Simultaneously, one shows a similar statement for $R \to_s S$. The interesting case is $\downarrow \lambda X R \mapsto_d \lambda x \downarrow (R[X := \uparrow x])$, where we have $(\downarrow \lambda X R)^* = \lambda x (R[X := \uparrow x])^*$. □

Lemma 4.1.2. $r \to_\beta s \implies r^* \to_{\underline{\beta}} s^*$.

Proof. Induction on reductions. The interesting case is that of an elementary reduction, where we have

$$((\lambda X R) S)^* = \lambda x (R[X := \uparrow x])^* S^*,$$

so that it suffices to show

$$(R[X := \uparrow x])^* [x := S^*] = (R[X := S])^*$$

for new x. This is an easy induction on R. □

Theorem 4.1.3. $[\![\underline{t}]\!]^* = \underline{t}$.

Proof. Induction on $\underline{t}$ using

$$[\![\lambda x \underline{t}]\!]^* = (\lambda X [\![\underline{t}]\!]_{\uparrow, x := X})^* = \lambda x ([\![\underline{t}]\!]_{\uparrow, x := X}[X := \uparrow x])^* = \lambda x [\![\underline{t}]\!]^* = \lambda x \underline{t},$$

where the substitution property used in the third equation can be shown by a simple induction on $\underline{t}$. □

Corollary 4.1.4. $\downarrow [\![\underline{t}]\!] \to^* \lambda \vec{x}(x \vec{R}) \Longrightarrow \underline{t} \to^*_{\underline{\beta}} \lambda \vec{x}(x \vec{R}^*)$.

Proof. By the proposition, syntactic reductions do not change the kernel of a term while by the lemma β-reductions translate to $\underline{\beta}$-reductions. Hence $(\lambda \vec{x}(x\vec{R}))^* = \lambda \vec{x}(x \vec{R}^*)$ is a reduct of $\downarrow [\![\underline{t}]\!]^* = [\![\underline{t}]\!]^*$ which by the theorem is equal to $\underline{t}$. □

The corollary implies in particular that all head normal forms returned by Normalization by Evaluation are correct.

4.2. *Normal form property*

We conclude the section by establishing that Normalization by Evaluation indeed only constructs normal forms. Note that this could also be achieved by an appropriate typing discipline (Danvy and Rhiger, 2001; Danvy et al., 2001).

During the evaluation of $\downarrow[\![\underline{r}]\!]$ a subexpression $\uparrow r$ should morally just contain Λ_1-terms r of the form $x\vec{r}$, so that the reduction $\downarrow\uparrow r \to_d r$ makes sense and leads to normal forms, only. Also all such terms r should be $\underline{\beta}$-normal. These restrictions are formalized in the following

Definition 4.2.1. The set of *honest terms* H is given inductively by the following simultaneous grammars.

$$\begin{array}{rcll} H \cap \Lambda_2 \ni & R, S & ::= & X \mid RS \mid \lambda XR \mid \uparrow(x\vec{r}), \\ H \cap \Lambda_1 \ni & r, s & ::= & x\vec{r} \qquad \mid \lambda xr \quad \mid \downarrow R. \end{array}$$

Obviously, $H \subset \mathrm{NF}_{\underline{\beta}}$, as desired. The goal is now to prove that honesty is an invariant during the rewriting of $\downarrow[\![\underline{r}]\!]$ and thus of Normalization by Evaluation.

Example 4.2.2. $[\![\underline{r}]\!] \in H$.

Lemma 4.2.3. $H \ni r \to s \implies s \in H$.

Proof. The claim is shown simultaneously with a similar assertion for Λ_2-terms, using induction on $\to$. It is immediate that H is closed under substitution for meta-variables and therefore under β-reduction. For the reduction $(\uparrow r)S \mapsto_a \uparrow(r\downarrow S)$ note that r has to be of the form $x\vec{r}$ with $\vec{r} \in H \cap \Lambda_1$, hence $x\vec{r}\downarrow S \in H \cap \Lambda_1$ and therefore $\uparrow(x\vec{r}\downarrow S) \in H \cap \Lambda_2$. The case of a reduction $\downarrow\uparrow r \mapsto_d r$ is straightforward. For the remaining reduction $\downarrow\lambda XR \mapsto_d \lambda x\downarrow(R[X := \uparrow x])$ we again use the fact that H is closed under the substitution $[X := \uparrow x]$.

For the induction step, the only interesting case is $\uparrow r \to \uparrow s$ with $r \to s$. Then r is of the form $x\vec{r}$ and the reduction has to be of the form $x\vec{r} \to x\vec{s}$, so we can use the induction hypothesis. □

Corollary 4.2.4. $\downarrow[\![\underline{r}]\!] \to^* s \implies s \in \mathrm{NF}_{\underline{\beta}}$.

Proof. By the example $[\![\underline{t}]\!] \in H$, hence $\downarrow[\![\underline{t}]\!] \in H$ and the lemma applies. □

In particular, we obtain $\downarrow[\![\underline{r}]\!] \to^* \underline{s} \implies \underline{s} \in \mathrm{NF}_{\underline{\beta}}$. Thus all results in Λ computed by Normalization by Evaluation are $\underline{\beta}$-normal.

5. Types and η-Expansion

In this section we fix a simple type discipline for Λ, introduce η-expansion and prove the combination with $\underline{\beta}$-reduction strongly normalizing and confluent.

5.1. *Types and type assignment*

(Simple) Types ρ, σ, τ are generated from basic types ι by $\rho \to \sigma$. Fixing a unique assignment $x\colon \rho$ of types to variable symbols such that for each type infinitely many variables exist, we can determine the typable terms and their unique type by the following rules:

$$\frac{\underline{r} : \rho \to \sigma \qquad \underline{s} : \rho}{\underline{r}\,\underline{s} : \sigma} \qquad \frac{\underline{r} : \sigma \qquad x : \rho}{\lambda x \underline{r} : \rho \to \sigma}$$

We will decorate (sub-)terms with types in superscripts (as in $\underline{r}^{\rho}\underline{s}$) in order to signify that they are typable and get the respective type. Λ^{ρ} denotes the set of terms of type ρ.

For the rest of this section we restrict our focus (and all quantifiers) to typable terms.

5.2. η*-expansion*

η-expansion has first been formally studied by Mints (1992). We refer to Jay's exposition (1995) for a discussion of the relevance of η-expansion and di Cosmo's slides (1996) for an overview of the literature.

Definition 5.2.1. A term is *neutral*, if it is not an abstraction.η*-expansion* is defined by the following elementary reduction rule.

$$\underline{r}^{\rho \to \sigma} \mapsto_{\eta\Uparrow} \lambda x^{\rho}(\underline{r}x), \qquad x \text{ new, } \underline{r} \text{ neutral.}$$

Clearly, η-expansion makes sense only in non-applicative contexts. Therefore the notion of term closure has to be modified for $\to_{\eta\Uparrow}$. We define $\to_{\eta\Uparrow}$ to be the least relation extending $\mapsto_{\eta\Uparrow}$ which is closed under

$$\frac{\underline{r} \to_{\eta\Uparrow} \underline{r}'}{(\lambda x \underline{r})\vec{\underline{s}} \to_{\eta\Uparrow} (\lambda x \underline{r}')\vec{\underline{s}}} \qquad \frac{\underline{s} \to_{\eta\Uparrow} \underline{s}'}{\underline{r}\,\underline{s}\vec{\underline{t}} \to_{\eta\Uparrow} \underline{r}\,\underline{s}'\vec{\underline{t}}}$$

NOTE that strong normalization for $\to_{\beta\eta\Uparrow} := \to_{\underline{\beta}} \cup \to_{\eta\Uparrow}$ is not obvious: although the definition of $\to_{\eta\Uparrow}$ prevents immediate creation of $\underline{\beta}$-redexes, an expansion might provoke new $\underline{\beta}$-reductions in the future, as in

$$(\lambda x \lambda y \underline{r})x \mapsto_{\eta\Uparrow} \lambda y((\lambda x \lambda y \underline{r})xy) \to_{\underline{\beta}} \lambda y((\lambda y \underline{r})y) \to_{\underline{\beta}} \lambda y \underline{r}.$$

Various proofs of strong normalization have been published (Akama, 1993; Dougherty, 1993; Cosmo and Kesner, 1994; Jay and Ghani, 1995). In this section we give a short and perspicuous strong normalization proof, following Akama's idea of simulating β-reductions on the $\eta\Uparrow$-normal form $r^{\eta\Uparrow}$ of a term r. The starting point will be the following inductive characterization of $\eta\Uparrow$-normal forms.

$$\begin{array}{lcll}
\mathrm{NF}_{\eta\Uparrow} & \ni \ \underline{r} & ::= & (x\vec{\underline{r}})^{\iota} \mid \lambda x \underline{r} \mid ((\lambda x \underline{r})\underline{s}\vec{\underline{s}})^{\iota} \\
\mathrm{NF}_{\underline{\beta}\eta\Uparrow} & \ni \ \underline{r} & ::= & (x\vec{\underline{r}})^{\iota} \mid \lambda x \underline{r}
\end{array}$$

5.3. *Head expansion*

We define a term $\eta_\rho \underline{r}^\rho \in \Lambda^\rho$ and the size $\mu_\rho \in \mathbb{N}$ of ρ by recursion on ρ:

$$\begin{array}{llll} \eta_\iota\, \underline{r} := \underline{r}, & \eta_{\rho\to\sigma}\, \underline{r} := \lambda x^\rho \eta_\sigma(\underline{r}\eta_\rho x), & x \text{ new}, \\ \mu_\iota \;\; := 0, & \mu_{\rho\to\sigma} \;\; := 1 + \mu_\sigma + \mu_\rho. \end{array}$$

Write $\eta\, \underline{r}^\rho$ for $\eta_\rho\, \underline{r}$. The following facts are easily verified.

(1) η preserves $\underline{\beta}$-reduction and substitution.[8]

(2) $\underline{r}^\rho \to_{\eta\Uparrow}^{\mu_\rho} \eta_\rho\, \underline{r}$ if $\underline{r}$ is neutral.
  [Induction on ρ. $\underline{r}^{\rho\to\sigma} \to_{\eta\Uparrow} \lambda x^\rho(\underline{r}x) \overset{\mathrm{IH}}{\to}{}_{\eta\Uparrow}^{\mu_\rho} \lambda x(\underline{r}\,\eta_\rho x) \overset{\mathrm{IH}}{\to}{}_{\eta\Uparrow}^{\mu_\sigma} \lambda x \eta_\sigma(\underline{r}\,\eta_\rho x)$.]

(3) If $\vec{\underline{r}}, \underline{r}, \underline{s}$ are in $\mathrm{NF}_{\eta\Uparrow}$ then so are $\eta_\rho(x\vec{\underline{r}}\,)$ and $\eta_\rho((\lambda x\underline{r})\underline{s}\vec{\underline{r}}\,)$.
  [Induction on ρ. $\eta_{\rho\to\sigma}(x\vec{\underline{r}}\,) = \lambda x^\rho \eta_\sigma(x\vec{\underline{r}}\,\eta_\rho x) \in \mathrm{NF}_{\eta\Uparrow}$ by IH_ρ and IH_σ.]

(4) $(\eta\underline{r})\vec{\underline{s}} \to_{\underline{\beta}}^* \eta(\underline{r}\,\eta\vec{\underline{s}}\,)$.
  [Induction on $\vec{\underline{s}}$. $(\eta_{\rho\to\sigma}\underline{r})\underline{s}\vec{\underline{s}} = (\lambda x\eta(\underline{r}\,\eta x))\underline{s}\vec{\underline{s}} \to_{\underline{\beta}} \eta(\underline{r}\,\eta\underline{s})\vec{\underline{s}}$.]

(5) $\eta\,\eta\underline{r}^\rho \to_{\underline{\beta}}^* \eta\underline{r}$.
  [Induction on ρ. $\eta\eta\underline{r}^{\rho\to\sigma} = \lambda x\eta((\eta\underline{r})\,\eta x) \overset{(4)}{\to}{}_{\underline{\beta}}^* \lambda x\eta\eta(\underline{r}\,\eta\eta x) \overset{\mathrm{IH}}{\to}{}_{\underline{\beta}}^* \lambda x\eta(\underline{r}\,\eta x)$.]

5.4. $\eta\Uparrow$*-normal form*

The $\eta\Uparrow$*-normal form* $\underline{r}^{\eta\Uparrow}$ of a term $\underline{r}^\rho$ is defined by recursion along the inductive term characterization of subsection 1.4.[9]

$$\begin{array}{llll} (x\vec{\underline{r}}\,)^{\eta\Uparrow} & := \eta(x\vec{\underline{r}}^{\,\eta\Uparrow}\,), & \#_{\eta\Uparrow}(x\vec{\underline{r}}\,) & := \mu_\rho + \sum \#_{\eta\Uparrow}\vec{\underline{r}}, \\ (\lambda x\underline{r})^{\eta\Uparrow} & := \lambda x\, \underline{r}^{\eta\Uparrow}, & \#_{\eta\Uparrow}\lambda x\underline{r} & := \#_{\eta\Uparrow}\underline{r}, \\ ((\lambda x\underline{r})\underline{s}\vec{\underline{s}}\,)^{\eta\Uparrow} & := \eta((\lambda x\, \underline{r}^{\eta\Uparrow})\, \underline{s}^{\eta\Uparrow}\, \vec{\underline{s}}^{\,\eta\Uparrow}\,), & \#_{\eta\Uparrow}((\lambda x\underline{r})\underline{s}\vec{\underline{s}}\,) & := \mu_\rho + \sum \#_{\eta\Uparrow}(\underline{r}, \underline{s}, \vec{\underline{s}}\,). \end{array}$$

Proposition 5.4.1 (Strong normalization and confluence for $\to_{\eta\Uparrow}$).

(6) $\underline{r} \to_{\eta\Uparrow}^{\#_{\eta\Uparrow}\underline{r}} \underline{r}^{\eta\Uparrow} \in \mathrm{NF}_{\eta\Uparrow}$,

(7) $\underline{r} \in \mathrm{NF}_{\eta\Uparrow} \iff \underline{r}^{\eta\Uparrow} = \underline{r} \iff \#_{\eta\Uparrow}\underline{r} = 0$,

(8) $\eta\underline{r}^{\eta\Uparrow} \to_{\underline{\beta}}^* \underline{r}^{\eta\Uparrow}$,

(9) $\underline{r}^{\eta\Uparrow}[x := \eta\, x] \to_{\underline{\beta}}^* \underline{r}^{\eta\Uparrow}$,

(10) $\eta(\underline{r}^{\eta\Uparrow}\, \vec{\underline{s}}^{\,\eta\Uparrow}\,) \to_{\underline{\beta}}^* (\underline{r}\,\vec{\underline{s}}\,)^{\eta\Uparrow}$,

(11) $\underline{r}^{\eta\Uparrow}[x := \underline{s}^{\eta\Uparrow}] \to_{\underline{\beta}}^* (\underline{r}[x := \underline{s}])^{\eta\Uparrow}$,

(12) $\underline{r} \to_{\eta\Uparrow} \underline{r}' \implies \underline{r}^{\eta\Uparrow} = \underline{r}'^{\eta\Uparrow}$ & $\#_{\eta\Uparrow}\underline{r} = \#_{\eta\Uparrow}\underline{r}' + 1$.

Proof. (6). $\underline{r}^{\eta\Uparrow} \in \mathrm{NF}_{\eta\Uparrow}$ by induction on $\underline{r}$ using (3). $\underline{r} \to_{\eta\Uparrow}^{\#_{\eta\Uparrow}\underline{r}} \underline{r}^{\eta\Uparrow}$ also by induction on $\underline{r}$ using (2).

(7). Use the above characterization of $\mathrm{NF}_{\eta\Uparrow}$ and (6).

(8) and (9) are proven by simultaneous induction on $\underline{r}$. (8). For neutral terms use (5).

[8] This will be used without further mentioning.

[9] Read $\#_{\eta\Uparrow}\vec{r}$, $\eta\vec{r}$ and $\vec{r}^{\,\eta\Uparrow}$ pointwise.

For an abstraction we compute

$$\begin{array}{rcll}
\eta(\lambda x\underline{r})^{\Uparrow} & = & \eta\,\lambda x\,\underline{r}^{\Uparrow} & \\
 & = & \lambda y\eta((\lambda x\,\underline{r}^{\Uparrow})\,\eta y) & \\
 & \to_{\underline{\beta}} & \lambda x\,\eta(\underline{r}^{\Uparrow}[x := \eta\,x]) & \\
 & \to_{\underline{\beta}}^{*} & \lambda x\,\eta\underline{r}^{\Uparrow} & \text{by IH(9)} \\
 & \to_{\underline{\beta}}^{*} & \lambda x\,\underline{r}^{\Uparrow} & \text{by IH(8).}
\end{array}$$

The only interesting case for (9) is $\underline{r} = x\underline{\vec{r}}$ where we have

$$\begin{array}{rcll}
(x\underline{\vec{r}})^{\Uparrow}[x := \eta x] & = & \eta(x\,\underline{\vec{r}}^{\Uparrow})[x := \eta x] & \\
 & = & \eta(\eta x\;\underline{\vec{r}}^{\Uparrow}\,[x := \eta x]) & \\
 & \to_{\underline{\beta}}^{*} & \eta(\eta x\;\underline{\vec{r}}^{\Uparrow}\,) & \text{by IH(9)} \\
 & \to_{\underline{\beta}}^{*} & \eta\,\eta(x\,\eta\,\underline{\vec{r}}^{\Uparrow}) & \text{by (4)} \\
 & \to_{\underline{\beta}}^{*} & \eta(x\,\eta\,\underline{\vec{r}}^{\Uparrow}) & \text{by (5)} \\
 & \to_{\underline{\beta}}^{*} & \eta(x\,\underline{\vec{r}}^{\Uparrow}) & \text{by IH(8)} \\
 & = & (x\underline{\vec{r}})^{\Uparrow}. &
\end{array}$$

(10) is shown by induction on $\underline{r}$.

$$\begin{array}{rcll}
\eta\,((x\underline{\vec{r}})^{\Uparrow}\;\underline{\vec{s}}^{\Uparrow}) & = & \eta(\eta(x\underline{\vec{r}}^{\Uparrow})\,\underline{\vec{s}}^{\Uparrow}) & \\
 & \to_{\underline{\beta}}^{*} & \eta\,\eta(x\;\underline{\vec{r}}^{\Uparrow}\;\eta\,\underline{\vec{s}}^{\Uparrow}) & \text{by (4)} \\
 & \to_{\underline{\beta}}^{*} & \eta\,\eta(x\,\underline{\vec{r}}^{\Uparrow}\;\underline{\vec{s}}^{\Uparrow}) & \text{by (8)} \\
 & \to_{\underline{\beta}}^{*} & \eta(x\,\underline{\vec{r}}^{\Uparrow}\;\underline{\vec{s}}^{\Uparrow}) & \text{by (5)} \\
 & = & (x\underline{\vec{r}}\underline{\vec{s}})^{\Uparrow}. & \\
\eta((\lambda x\underline{r})^{\Uparrow}\;\underline{\vec{s}}^{\Uparrow}) & = & \eta((\lambda x\underline{r}^{\Uparrow})\,\underline{\vec{s}}^{\Uparrow}) & \\
 & = & ((\lambda x\underline{r})\,\underline{\vec{s}})^{\Uparrow}. &
\end{array}$$

(11) is an easy induction on $\underline{r}$. We treat only the case $x\underline{\vec{r}}$ where (10) is needed.

$$\begin{array}{rcll}
(x\underline{\vec{r}})^{\Uparrow}[x := \underline{s}^{\Uparrow}] & = & \eta(\underline{s}^{\Uparrow}\;(\underline{\vec{r}}^{\Uparrow}[x := \underline{s}^{\Uparrow}])) & \\
 & \to_{\underline{\beta}}^{*} & \eta(\underline{s}^{\Uparrow}\;(\underline{\vec{r}}\,[x := \underline{s}])^{\Uparrow}) & \text{by IH} \\
 & \to_{\underline{\beta}}^{*} & (\underline{s}\;\vec{r}\,[x := \underline{s}])^{\Uparrow} & \text{by (10).}
\end{array}$$

(12). For an $\eta\!\Uparrow$-expansion $\underline{r} \mapsto_{\eta\Uparrow} \lambda x(\underline{r}x)$, r neutral, we treat the case $\underline{r} = x\underline{\vec{r}}$.

$$\begin{array}{rcl}
(x\underline{\vec{r}})^{\Uparrow} & = & \eta_{\rho\to\sigma}(x\,\underline{\vec{r}}^{\Uparrow}) \\
 & = & \lambda y^{\rho}\,\eta_{\sigma}(x\,\underline{\vec{r}}^{\Uparrow}\,\eta_{\rho}y) \\
 & = & (\lambda y(x\underline{\vec{r}}y))^{\Uparrow}. \\
\#_{\eta\Uparrow}(x\underline{\vec{r}})^{\rho\to\sigma} & = & \mu_{\rho\to\sigma} + \sum \#_{\eta\Uparrow}\underline{\vec{r}} \\
 & = & 1 + \mu_{\sigma} + \sum \#_{\eta\Uparrow}\underline{\vec{r}} + \mu_{\rho} \\
 & = & 1 + \mu_{\sigma} + \sum \#_{\eta\Uparrow}(\underline{\vec{r}}, y^{\rho}) \\
 & = & 1 + \#_{\eta\Uparrow}\lambda y^{\rho}(x\underline{\vec{r}}y).
\end{array}$$

The case $\underline{r} = (\lambda x\underline{s})\underline{t}\underline{\vec{t}}$ is analogous. The cases of inner reductions follow immediately from the induction hypothesis. ☐

Lemma 5.4.2 (Simulation). $\underline{r} \to_{\beta} \underline{s} \implies \underline{r}^{\Uparrow} \to_{\underline{\beta}}^{+} \underline{s}^{\Uparrow}$.

Proof. Induction on $\underline{r}$. We show only the interesting case of an elementary $\underline{\beta}$-reduction.

$$\begin{array}{rcll}
((\lambda x\underline{r})\underline{t})^{\eta\Uparrow} & = & \eta((\lambda x\,\underline{r}^{\eta\Uparrow})\underline{t}^{\eta\Uparrow}) & \\
 & \to_{\underline{\beta}} & \eta(\underline{r}^{\eta\Uparrow}[x := \underline{t}^{\eta\Uparrow}]) & \\
 & \to_{\underline{\beta}}^{*} & \eta(\underline{r}[x := \underline{t}])^{\eta\Uparrow} & \text{by (11)} \\
 & \to_{\underline{\beta}}^{*} & (\underline{r}[x := \underline{t}])^{\eta\Uparrow} & \text{by (8).}
\end{array}$$

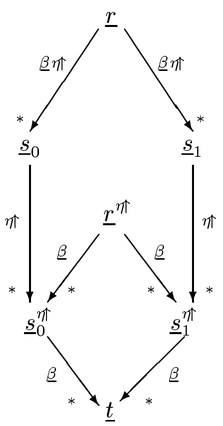
All other cases follow directly from the induction hypothesis. □

5.5. *Strong normalization for* $\to_{\beta\eta\Uparrow}$

Using the lemma we can simulate any $\underline{\beta}$-reduction on a term $\underline{r}$ by a positive number of $\underline{\beta}$-reductions on $\underline{r}^{\eta\Uparrow}$, while η-expansions leave $\underline{r}^{\eta\Uparrow}$ unchanged by (12). Thus strong normalizability of $\underline{r}$ follows by induction on $\to_{\underline{\beta}}$ and side induction on $\#_{\eta\Uparrow}\underline{r}$.

5.6. *Confluence for* $\to_{\beta\eta\Uparrow}$

See the picture. Assume $\underline{r}$ reduces to both $\underline{s}_0$ and $\underline{s}_1$. Then $\underline{\beta}$-simulation gives $\underline{r}^{\eta\Uparrow} \to_{\underline{\beta}}^{*} \underline{s}_i^{\eta\Uparrow}$ for each $\underline{s}_i$. Confluence for $\underline{\beta}$ yields a common $\underline{\beta}$-reduct $\underline{t}$ of $\underline{s}_0^{\eta\Uparrow}$ and $\underline{s}_1^{\eta\Uparrow}$. Since $\underline{s}_i \to_{\eta\Uparrow}^{*} \underline{s}_i^{\eta\Uparrow}$, we obtain $\underline{s}_i \to_{\beta\eta\Uparrow}^{*} t$.



6. Typed Normalization by Evaluation

In this section a type system for the two-level λ-calculus is defined and shown to imply strong normalization for $\Lambda_{1,2}$. The Normalization by Evaluation algorithm is properly adapted, so that it computes long $\beta\eta$-normal forms and can be compared to the usual type-directed Normalization by Evaluation approach (Danvy, 1998).

6.1. *Types*

In principle, we should assign types ρ to Λ_1-terms and meta-types $\boldsymbol{\rho}$ for Λ_2, using coercions to relate the two type hierarchies. For simplicity we identify them and arrive at the following typing calculus.

$$\frac{r : \sigma \qquad x : \rho}{\lambda x^{\rho} r : \rho \to \sigma} \qquad \frac{r : \rho \to \sigma \qquad s : \rho}{rs : \sigma} \qquad \frac{R : \rho}{\downarrow R : \rho}$$

$$\frac{R : \sigma \qquad X : \rho}{\lambda X R : \rho \to \sigma} \qquad \frac{R : \rho \to \sigma \qquad S : \rho}{RS : \sigma} \qquad \frac{r : \rho}{\uparrow r : \rho}$$

Let Λ_1^{ρ} (and Λ_2^{ρ}) denote the set of terms of Λ_1 (Λ_2, respectively) of type ρ, given the fixed typing for variables. For the rest of this section let all mentioned terms be typable.

6.2. *Adding η-expansion*

The function η_ρ can be extended to Λ_1, for it does not analyze the shape of its argument. $\mapsto_d$ is changed to

$$\downarrow\uparrow r^\rho \mapsto_d \eta_\rho r.$$

By mere inspection of elementary reduction rules, subject reduction holds.

Proposition 6.2.1. $\underline{r} \in \mathrm{NF}_{\underline{\beta}} \implies \downarrow[\![\underline{r}]\!] \to_s^* \underline{r}^{\Uparrow}$.

Proof. Induction on NF_β.
Case $\lambda x\underline{r}$. Again using example 2.2.1.
Case $x\vec{\underline{r}}$.

$$\begin{array}{rcll} \downarrow[\![x\vec{\underline{r}}]\!] & = & \downarrow((\uparrow x)[\![\vec{\underline{r}}]\!]) & \\ & \to_a^* & \downarrow\uparrow(x\downarrow[\![\vec{\underline{r}}]\!]) & \\ & \to_s^* & \downarrow\uparrow(x\vec{\underline{r}}^{\Uparrow}) & \text{by IH} \\ & \to_d & \eta(x\vec{\underline{r}}^{\Uparrow}) & \\ & = & (x\vec{r})^{\Uparrow}. & \end{array}$$

□

6.3. *Strong normalization for $\to$*

Lemma 6.3.1. $\to_s$ is strongly normalizing.

Proof. We show strong normalizability of R and r by simultaneous induction on the number of meta-applications and meta-abstractions in the terms, with side induction on the number of occurrences of $\downarrow$. Obviously, $(\uparrow r)S \mapsto_a \uparrow(r\downarrow S)$ and the reduction $\downarrow\lambda XR \mapsto_d \lambda x\downarrow R[X := \uparrow x]$ reduce the first, while $\downarrow\uparrow r \mapsto_d \eta r$ decreases the second measure. □

Proposition 6.3.2 (Commutation). $\to_s\to_\beta \;\subseteq\; \to_\beta\to_s^*$.

Proof. Induction on $\to_s$. The elementary reduction rules simply commute with a subsequent β-reduction. Of the remaining cases only the subcase of a following β-reduction $(\lambda XR)S \to_s (\lambda XR')S' \mapsto_\beta R'[X := S']$ is interesting, where we reduce $(\lambda XR)S \mapsto_\beta R[X := S] \to_s^* R'[X := S']$, using parallel substitutivity of $\to_s^*$. □

Corollary 6.3.3. $\to$ is strongly normalizing.

Proof. $\to_\beta$ is shown to be strongly normalizing by a trivial adaptation of the usual proof for typed Λ. Thus we can use the fact that the union of two strongly normalizing reduction relations that commute is again strongly normalizing. □

6.4. *Type-directed Normalization by Evaluation*

We define the functions

$$\uparrow_\rho : \Lambda_1^\rho \to \Lambda_2^\rho \qquad \text{and} \qquad \downarrow_\rho : \Lambda_2^\rho \to \Lambda_1^\rho$$

by simultaneous recursion on ρ:

$$\begin{array}{lll}
\downarrow_{\rho\to\sigma} R & := & \lambda x \downarrow_\sigma (R \uparrow_\rho x), \qquad x \text{ new}, \\
\uparrow_{\rho\to\sigma} r & := & \lambda X \uparrow_\sigma (r \downarrow_\rho X), \qquad X \text{ new}, \\
\downarrow_\iota R & := & \begin{cases} r & \text{if } R = \uparrow r, \\ \downarrow R & \text{otherwise.} \end{cases} \\
\uparrow_\iota r & := & \uparrow r.
\end{array}$$

A trivial simultaneous induction on ρ shows that β-reduction and substitution into meta-variables are preserved by $\downarrow_\rho$ and $\uparrow_\rho$.

Proposition 6.4.1. $\downarrow_\rho \uparrow_\rho r \to_\beta^* \eta_\rho r$.

Proof. Induction on ρ. **Case** ι. Simple. **Case** $\rho \to \sigma$.

$$\begin{array}{lll}
\downarrow_{\rho\to\sigma} \uparrow_{\rho\to\sigma} r & = & \lambda x \downarrow_\sigma ((\lambda X \uparrow_\sigma (r \downarrow_\rho X)) \uparrow_\rho x) \\
& \to_\beta & \lambda x \downarrow_\sigma \uparrow_\sigma (r \downarrow_\rho \uparrow_\rho x) \\
& \to_\beta^* & \lambda x \eta_\sigma (r \eta_\rho x) \qquad \text{by IH} \\
& = & \eta_{\rho\to\sigma} r.
\end{array}$$

□

6.5. *Simulation*

With the functions $\uparrow_\rho$ and $\downarrow_\rho$ we can simulate $\to_s$-reductions by $\to_\beta$:

$$\begin{array}{l}
\downarrow_{\rho\to\sigma} \lambda X R = \lambda x \downarrow_\sigma ((\lambda X R) \uparrow_\rho x) \to_\beta \lambda x \downarrow_\sigma R[X := \uparrow_\rho x]. \\
\downarrow_\rho \uparrow_\rho r \to_\beta^* \eta_\rho r \qquad \text{by the proposition.} \\
(\uparrow_{\rho\to\sigma} r) S = (\lambda X \uparrow_\sigma (r \downarrow_\rho X)) S \to_\beta \uparrow_\sigma (r \downarrow_\rho S).
\end{array}$$

Thus $\downarrow_\rho$ and $\uparrow_\rho$ emulate the behavior of $\downarrow$ and $\uparrow$. In other words, type-directed Normalization by Evaluation is a model of our calculus.

7. Implementation in `Haskell`

This section describes an example implementation of the normalization algorithm in `Haskell`.

7.1. *Terms of* Λ

We implement the calculus Λ with de Bruijn-terms in order to avoid all difficulties of bound variable renaming. Thus variables are represented by numbers, corresponding to their position in the list of free variables, where abstraction always affects the free variable with the lowest number.

Relative to a unique enumeration of free variables, λ-terms are in a one-to-one correspondence with de Bruijn-terms. If, for instance, x is the 5th variable in such an enumeration, then the λ-term $x\lambda y\lambda z(xzy)$ would be respresented by $4\lambda\lambda(6\,0\,1)$.

The implementation of de Bruijn-terms in `Haskell` is straightforward:

```
data Term = Var Integer | App Term Term | Abs Term
            deriving Show
```

Programming with de Bruijn-terms involves lifting and is notoriously laborious to work with. As a simple interface to `Term` that allows to access the n-th abstracted variable, we provide the type `TERM`.

```
type TERM = Integer -> Term
```

Applying an object of type `TERM` to a number `n` should produce the underlying `Term` with the free variables lifted by `n` (the *de Bruijn-* or *binding level*).

```
inspect :: TERM -> Term
inspect f = f 0
```

Remark. In a proof of correctness of the implementation, this intuition about `TERM` would have to be formalized as an invariant: the semantics of an object f of type `TERM` is $f0$ and fn is the nth lifting of $f0$.

The `k`th free variable is obtained by

```
freevar :: Integer -> TERM
freevar k = \ n -> Var (n + k)
```

Application combines two `TERM`s at the same binding level

```
apply :: TERM -> TERM -> TERM
apply r s = \ n -> App (r n) (s n)
```

while abstraction increases the binding level of its argument by one.

```
abstract :: TERM -> TERM
abstract r = \ n -> Abs (r (n + 1))
```

At binding level n, we can address each of the n bound variables by

```
boundvar :: Integer -> TERM
boundvar k = \ n -> Var (n - k - 1)
```

For instance, `abstract (abstract (boundvar 0))` corresponds to the term $\lambda x\lambda yx$ (or $\lambda\lambda 1$ in de Bruijn-notation):

```
Main> inspect (abstract (abstract (boundvar 1)))
Abs (Abs (Var 0))
Main> inspect (abstract (abstract (boundvar 0)))
Abs (Abs (Var 1))
```

Note that `boundvar k n` only works correctly if `k<n`.

7.2. *Terms of* Λ_2

The conversions $\mapsto_\beta$, $\mapsto_a$ and $\mapsto_d$ of the meta-level are modeled by function definitions of meta-application and $\downarrow$ in `Haskell`. The head constructor of a β- or a-redex is a meta-application. Similarly d-redexes are identified by the head constructor $\downarrow$. Meta-variables are modeled by `Haskell`-variables. Of the remaining term constructors, meta-abstraction will be represented by `Haskell`'s functional abstraction mechanism; the $\uparrow$ constructor is named `Up`:

```
data LAM = Up TERM | ABS (LAM -> LAM)
```

The implementation of three of the conversion rules is straightforward:

```
app :: LAM -> LAM -> LAM
app (ABS f) r = f r                     -- the beta-rule
app (Up r)  s = Up (apply r (down s))  -- the a-rule

down :: LAM -> TERM                     -- the first d-rule
down (Up r)   = r
```

By the basic soundness property of `Haskell`'s evaluation mechanism, these clauses model the respective rewriting rules.

The remaining conversion rule $\downarrow\lambda XR \mapsto_d \lambda x{\downarrow}R[X := \uparrow x]$ requires some thought: direct translation leads to

```
down (ABS f) = abstract (down (f (Up ?)))
```

Note that `abstract` automatically lifts all variables of its argument, so that the variable 0 becomes free. We would thus like to use 0 at the position of `?`, but `f` might actually introduce new abstractions, thus increasing the binding level and changing the role of 0. Therefore we need to refer to the absolute position of the bound variable introduced by `abstract` in the list of abstractions by η-expansion:

```
down (ABS f) = \ n -> abstract (down (f (Up (boundvar n)))) n
```

7.3. *Evaluation*

A valuation ξ maps object variables (implemented as `Integer`) to meta-level terms.

```
type Valuation = Integer -> LAM
```

Modification $\xi, x := R$ of a valuation ξ is implemented only for the first variable:

```
comma :: Valuation -> LAM -> Valuation
comma xi r n = if n == 0 then r else xi (n - 1)
```

Now we can define the clauses for evaluation:

```
eval :: Term -> Valuation -> LAM
eval (Var n)   xi = xi n
eval (Abs r)   xi = ABS (\ a -> eval r (xi `comma` a))
eval (App r s) xi = (eval r xi) `app` (eval s xi)
```

Normalization by evaluation is defined by $\downarrow[\![\underline{r}]\!]_{\uparrow}$.

```
nbe r = inspect (down (eval r (Up . freevar)))
```

7.4. *Examples*

Standard combinators and $\omega := \lambda x(xx)$, $\Omega := \omega\omega$ are given by

```
k = Abs (Abs (Var 1))
s = Abs (Abs (Abs (Var 2 `App` (Var 0) `App`
                   (Var 1 `App` (Var 0)))))
i = s `App` k `App` k
omega = Abs (Var 0 `App` (Var 0))
oomega = omega `App` omega         -- not terminating
```

Here are some instructive examples

```
Main> nbe k
Abs (Abs (Var 1))
Main> nbe i
Abs (Var 0)
Main> nbe (k `App` i `App` oomega)
Abs (Var 0)
```

7.5. *Böhm Trees*

Theorem 3.5 states that our approach can be used to calculate Böhm trees as well. The fixpoint combinator

$$Y := \lambda f\,((\lambda x\,(f(xx)))\,\lambda x\,(f(xx)))$$

has the infinite Böhm tree $\lambda f(f(f(\ldots)))$ (Barendregt, 1977). The definition of Y reads

```
y = Abs  ((Abs (Var 1 `App` (Var 0 `App` Var 0))) `App`
           (Abs (Var 1 `App` (Var 0 `App` Var 0))))
```

Since `Haskell` prints the calculated part of a result as it proceeds, we can have a look at the beginning of the Böhm tree of Y.

```
Main> nbe y
Abs (App (Var 0) (App (Var 0) (App (Var 0) (App (Var 0) ...
```

In general, a Böhm tree might have undefined (diverging) leaves, although neighboring branches are still worthwhile looking at. So we define a function `cut` that replaces a set of subtrees of its argument (as identified by a list of `paths`) by a canonical tree, for which we use the illegal variable -1

```
cutvalue = Var (-1)
```

The auxiliary function `shorten c paths` yields the tail of those elements of `paths` that begin with `c`.

```
shortenpath c (a:as) | c == a  = [as]
shortenpath _  _               = []    -- otherwise

shorten c paths =  paths >>= shortenpath c
```

The function `cut` is defined by

```
cut  _         paths
        | "" `elem` paths = cutvalue
-- if the empty path has to be removed nothing remains
```

```
cut (Var n)   paths   = Var n
cut (Abs t)   paths   = Abs (cut t (shorten '0' paths))
cut (App r s) paths   = App (cut r (shorten '0' paths))
                            (cut s (shorten '1' paths))
```

This allows to use the above examples *without* eventually interrupting the interpreter.

```
Main> cut (nbe y) ["01111111111"]
Abs (App (Var 0) (App (Var 0) (App (Var 0) (App (Var 0) (App
(Var 0) (App (Var 0) (App (Var 0) (App (Var 0) (App (Var 0)
(App (Var 0) (Var (-1))))))))))))
```

Due to the undecidability of the halting problem, the undefined paths of a Böhm tree cannot be identified, so that this information has to be provided by hand. Consider the pair and swap operators

```
pair = Abs (Abs (Abs (Var 0 `App` Var 2 `App` Var 1)))
swap = Abs (Abs (Abs (Var 0 `App` Var 1 `App` Var 2)))
```

We can use the heuristics that the Böhm tree is undefined, if the runtime is too long.

```
Main> nbe (pair `App` i `App` oomega)
Abs (App (App (Var 0) (Abs (Var 0))) {Interrupted!}

Main> cut (nbe (pair `App` i `App` oomega)) ["01"]
Abs (App (App (Var 0) (Abs (Var 0))) (Var (-1)))

Main> nbe (pair `App` i `App` oomega `App` swap)
Abs (App (App (Var 0) {Interrupted!}

Main> cut (nbe (pair `App` i `App` oomega `App` swap)) ["001"]
Abs (App (App (Var 0) (Var (-1))) (Abs (Var 0)))
```

References

Akama, Y. (1993). On Mints' reductions for ccc-calculus. In (Bezem and Groote, 1993), pages 1–12.

Altenkirch, T., Hofmann, M., and Streicher, T. (1995). Categorical reconstruction of a reduction free normalization proof. In Pitt, D. H., Rydeheard, D. E., and Johnstone, P., editors, *Proc. 6^{th} CTCS (Cambridge)*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer.

Altenkirch, T., Hofmann, M., and Streicher, T. (1996). Reduction-free normalisation for a polymorphic system. In *Proc. 11^{th} LICS (New Brunswick)*, pages 98–106. IEEE Computer Society Press.

Barendregt, H. (1977). The type free lambda calculus. In Barwise, J., editor, *Handbook of Mathematical Logic*, chapter D.7, pages 1091–1132. North-Holland.

Berger, U. (1993). Program extraction from normalization proofs. In (Bezem and Groote, 1993), pages 91–106.

Berger, U., Eberl, M., and Schwichtenberg, H. (1998). Normalization by evaluation. In Möller,

B. and Tucker, J., editors, *Prospects for Hardware Foundations*, number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer.

Berger, U. and Schwichtenberg, H. (1991). An inverse of the evaluation functional for typed λ–calculus. In Vemuri, R., editor, *Proc. 6^{th} LICS (Amsterdam)*, pages 203–211. IEEE Computer Science Press.

Bezem, M. and Groote, J., editors (1993). *Proc. 1^{st} TLCA 1993 (Utrecht)*, volume 664 of *Lecture Notes in Computer Science*. Springer.

Bruijn, N. d. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicae*, 34:381–392.

Coquand, T. and Dybjer, P. (1997). Intuitionistic model constructions and normalization proofs. *Math. Structures in Computer Science*, 7(1):75–94.

Cosmo, R. d. (1996). A brief history of rewriting with extensionality. Presented at the International Summer School on Type Theory and Term Rewriting (Glasgow). Available online.

Cosmo, R. d. and Kesner, D. (1994). Simulating expansions without expansions. *Math. Structures in Computer Science*, 4(3):1–48.

Čubrić, D., Dybjer, P., and Scott, P. (1998). Normalization and the Yoneda embedding. *Math. Structures in Computer Science*, 8(2):153–192.

Danvy, O. (1998). Type-directed partial evaluation. Number 1706 in Lecture Notes in Computer Science, pages 367–411. Springer.

Danvy, O. and Rhiger, M. (2001). A simple take on typed abstract syntax in Haskell-like languages. In Kuchen, H. and Ueda, K., editors, *Proc. 5^{th} Symposium on Functional and Logic Programming*, volume 2024 of *Lecture Notes in Computer Science*, pages 343–358.

Danvy, O., Rhiger, M., and Rose, K. H. (2001). Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–80.

Dougherty, D. (1993). Some lambda calculi with categorial sums and products. In Kirchner, C., editor, *Proc. 5^{th} RTA 1993 (Montreal)*, volume 690 of *Lecture Notes in Computer Science*, pages 137–151. Springer.

Goldberg, M. (1996). Gödelization in the lambda calculus. Technical Report RS-96-5, BRICS, Denmark.

Goldberg, M. (2000). Gödelization in the lambda calculus. *Information Processing Letters*, 75(1–2):13–16.

Jay, B. and Ghani, N. (1995). The virtues of eta-expansion. *Journal of Functional Programming*, 5:135–154.

Joachimski, F. and Matthes, R. (2000). Standardization and confluence for a lambda calculus with generalized applications. In Bachmair, L., editor, *Proc. 11^{th} RTA (Norwich)*, volume 1833 of *Lecture Notes in Computer Science*, pages 141–155. Springer.

Mints, G. (1992). Proof theory and category theory. In *Selected Papers in Proof Theory*, Studies in Proof Theory, chapter 10, pages 157–182. Bibliopolis.

Mogensen, T. A. (1999). Gödelization in the untyped lambda-calculus. In Danvy, O., editor, *Proc. PEPM (San Antonio)*, Technical report BRICS-NS-99-1, University of Aarhus, pages 19–24, San Antonio, Texas.

Oostrom, V. v. (1997). Developing developments. *Theoretical Computer Science*, 175(1):159–181.

Vestergaard, R. (2001). The simple type theory of normalisation by evaluation. In Gramlich, B. and Lucas, S., editors, *Proc. of WRS-1*, volume 2001.2359 of *Technical report series, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia.*