Term rewriting for normalization by evaluation

Ulrich Berger, Matthias Eberl and Helmut Schwichtenberg*

March 29, 2001

Abstract

We extend normalization by evaluation (first presented in [5]) from the pure typed λ-calculus to general higher type term rewriting systems and prove its correctness w.r.t. a domain-theoretic model. We distinguish between computational rules and proper rewrite rules. The former is a rather restricted class of rules, which, however, allows for a more efficient implementation.

1 Introduction

It is well known that implementing normalization of λ-terms in the usual recursive fashion is quite inefficient. However, it is possible to compute the long normal form of a λ-term by evaluating it in an appropriate model (cf. [5]). When using for that purpose the built-in evaluation mechanism of e.g. SCHEME (a pure LISP dialect) one obtains an amazingly fast algorithm called "normalization by evaluation" or NbE for short. In the context of type-directed partial evaluation [8] it has been analyzed in what sense NbE is more efficient, and why; a punctual comparison between NbE and a naive, symbolic normalizer can be found in [4, section 5]. The essential idea is to find an inverse to evaluation, converting a semantic object into a syntactic term. This normalization procedure is used and tested in the proof system MINLOG developed in Munich (cf. [2]). – Notice, however, that once NbE is expressed in a functional programming language, the evaluation order of this language (call-by-value for SCHEME) determines the reduction order of NbE (applicative order for a call-by-value language). It is thus easy to defeat NbE in SCHEME by normalizing the application of a non-strict function to an expression that is expensive to normalize. For such a term, a symbolic normalizer following a normal order reduction strategy can easily be more efficient.

Obviously, for applications pure typed λ-terms are not sufficient; one clearly needs constants as well. In [4] NbE has been extended to term systems with higher order term rewrite rules. The present paper adds a distinction between

*The hospitality of the Mittag-Leffler Institute in the spring of 2001 is gratefully acknowledged.

what we call computational rules and (proper) rewrite rules; NbE seems to be much more efficient for the former than for the latter. In our implementation (in the MINLOG system) we therefore use computational rules whenever possible.

A related approach (using a glueing construction) is elaborated by T. COQUAND and P. DYBJER in [6]. Another related paper is T. ALTENKIRCH, M. HOFMANN and T. STREICHER [1]; there a cartesian closed category is defined which has the property that the interpretation of the simply typed lambda calculus in it yields the reduction-free normalization algorithm from [5], as well as its correctness. Moreover, O. DANVY (cf. e.g. [8]) has successfully used this algorithm (or more precisely its call-by-value counterpart) in the context of partial evaluation. A. FILINSKI [10] also treats NbE for an extension of the λ-calculus by constants, where non-termination is allowed. However, he does not consider constants whose meaning is only given operationally, i.e. by arbitrary rewrite rules. Therefore the normal proof technique employing the logical relation "the value of expression e in environment δ is a" is available in his case, whereas in ours it is more convenient to follow a different approach, via an appropriate inductive generation of the reducibility relation.

Why should one be interested in the correctness of NbE for general rewrite rules, where neither termination nor even confluence is assumed? One reason is that in an interactive proof development system (MINLOG in our case) it is convenient not having to deal explicity with equality axioms, but rather to identify terms with the same normal form, modulo a given set of rewrite rules. Then an efficient normalization algorithm like NbE to test for equality clearly is useful. However, one does not want to have the obligation to prove termination and confluence of the whole set of rewrite rules whenever a new one is added.

The aim of the present paper is to develop the theory of normalization by evaluation from scratch, up to and including (some generalizations of) GÖDEL's system T of higher order primitive recursion. In fact, we will treat almost arbitrary rewrite systems.

Let us begin with a short explanation of the essence of the method for normalizing typed λ-terms by means of an evaluation procedure of some functional programming language such as SCHEME. For simplicity we return to the simplest case, simply typed λ-calculus without constants.

Simple types are built from ground types τ by $\rho \to \sigma$ (later also products $\rho \times \sigma$ will be included). The set Λ of terms is given by x^σ, $(\lambda x^\rho M^\sigma)^{\rho\to\sigma}$, $(M^{\rho\to\sigma} N^\rho)^\sigma$; let Λ_ρ denote the set of all terms of type ρ. The set LNF of terms in long normal form (i.e. normal w.r.t. β-reduction and η-expansion) is defined inductively by $(xM_1 \dots M_n)^\tau$, $\lambda x M$ (we abbreviate $xM_1 \dots M_n$ by $x\boldsymbol{M}$ and similar a list $M_1 \dots M_n$ by $\boldsymbol{M}$). By $\mathsf{nf}(M)$ we denote the long normal form of M, i.e. the unique term in long normal form $\beta\eta$-equal to M.

Now we have to choose our model. A simple solution is to take terms of ground type as ground type objects, and all functions as possible function type objects:

$$[\![\tau]\!] := \Lambda_\tau, \quad [\![\rho \to \sigma]\!] := [\![\sigma]\!]^{[\![\rho]\!]} \quad \text{(the full function space)}.$$

It is crucial that all terms (of ground type) are present, not just the closed ones.

Next we need an assignment $\uparrow$ lifting a variable to an object, and a function $\downarrow$ giving us a normal term from an object. They should meet the following condition, which might be called "correctness of normalization by evaluation"

$$\downarrow(\llbracket M \rrbracket_{\uparrow}) = \mathsf{nf}(M),$$

where $\llbracket M^{\rho} \rrbracket_{\uparrow} \in \llbracket \rho \rrbracket$ denotes the value of M under the assignment $\uparrow$. Two such functions $\downarrow$ and $\uparrow$ can be defined simultaneously, by induction on the type. It is convenient to define $\uparrow$ on all terms (not just on variables). Hence for every type ρ we define $\downarrow_{\rho} \colon \llbracket \rho \rrbracket \to \Lambda_{\rho}$ and $\uparrow_{\rho} \colon \Lambda_{\rho} \to \llbracket \rho \rrbracket$ (called reify and reflect) by

$$\begin{aligned} &\downarrow_{\tau}(M) := M, & &\uparrow_{\tau}(M) := M, \\ &\downarrow_{\rho\to\sigma}(a) := \lambda x \downarrow_{\sigma}(a(\uparrow_{\rho}(x))) \quad \text{“}x \text{ new”}, & &\uparrow_{\rho\to\sigma}(M)(a) := \uparrow_{\sigma}(M \downarrow_{\rho}(a)). \end{aligned}$$

Here a little difficulty appears: what does it mean that x is new? This clearly is not a problem for an implementation, where we have an operational understanding and may use something like **gensym**, but it is for a mathematical model. We will solve this problem by slightly modifying the model and defining $\llbracket \tau \rrbracket$ to be the set of families of terms of type τ (instead of single terms) and setting $\downarrow_{\rho\to\sigma}(a)(k) := \lambda x_k (\downarrow_{\sigma}(a(\uparrow_{\rho}(x_k^{\infty})))(k+1))$, where x_k^{∞} is the constant family x_k. The definition of $\uparrow_{\rho\to\sigma}$ has to be modified accordingly. This idea corresponds to a representation of terms in the style of de Bruijn [9]. An advantage of this approach is that the NbE program is purely functional and hence can be verified relatively easily. If side effects were involved the verification would be much more complicated.

The proof of correctness is easy (ignoring the problem with the "new variable"): Since for the typed lambda calculus without constants we have preservation of values, i.e. $\llbracket M \rrbracket_{\xi} = \llbracket \mathsf{nf}(M) \rrbracket_{\xi}$ for all terms M and environments ξ, we only have to verify $\downarrow(\llbracket N \rrbracket_{\uparrow}) = N$ for terms N in long normal form, which is straightforward, by induction on N:

Case $x^{\rho\to\tau} N^{\rho}$ (w.l.o.g.)

$$\downarrow_{\tau}(\llbracket xN \rrbracket_{\uparrow}) = \uparrow_{\rho\to\tau}(x)(\llbracket N \rrbracket_{\uparrow}) = \uparrow_{\tau}(x \downarrow_{\rho}(\llbracket N \rrbracket_{\uparrow})) = xN$$

Case $\lambda y N$

$$\begin{aligned} \downarrow_{\rho\to\sigma}(\llbracket \lambda y N \rrbracket_{\uparrow}) &= \lambda x \downarrow_{\sigma}(\llbracket \lambda y N \rrbracket_{\uparrow}(\uparrow_{\rho}(x))) && x \text{ new} \\ &= \lambda x \downarrow_{\sigma}(\llbracket N_y[x] \rrbracket_{\uparrow}) \\ &= \lambda x N_y[x] && \text{by IH} \\ &=_{\alpha} \lambda y N \end{aligned}$$

Notice that this is a correctness proof in the style of [5]. The situation is different when we add constants together with rewrite rules, since then preservation of values (in our model) is false in general (cf. examples 20 and 19 below). However, correctness of normalization by evaluation still holds, but needs to be proven by a different method. It might be worth noting that in the special case where no rewrite or computation rules are present our proof below boils down to the simple correctness proof sketched above.

The structure of the paper is as follows. In section 2 we present the simply typed λ-calculus with constants and pairing and give some examples of higher order rewrite systems. We also introduce the distinction between computational and (proper) rewrite rules. Then we inductively define a relation $M \longrightarrow Q$, with the intended meaning that M is normalizable with long normal form Q, and prove in section 3.6 the correctness of normalization by evaluation by showing that $M \longrightarrow Q$ (essentially) implies $\downarrow(\llbracket M \rrbracket_{\uparrow}) = Q$. Hence the mapping $M \mapsto \downarrow(\llbracket M \rrbracket_{\uparrow})$ is a normalization function. In order to define the semantics $\llbracket M \rrbracket$ of a term M properly we use domain theory. This is described briefly in section 3.1.

Note that we prove correctness of NbE w.r.t. a denotational semantics, but do not attempt to prove operational correctness, i.e. the fact that the functional program formalizing NbE when called with a term M such that $M \longrightarrow Q$ will terminate with Q as output. In order to obtain operational correctness from denotational correctness one needs a suitable adequacy result á la PLOTKIN [13] relating the denotational and the operational semantics. PLOTKIN's result cannot be applied here because it refers to a call-by-name operational semantics, whereas we are interested in a call-by-value semantics in order to obtain a correctness result for our implementation of NbE in the call-by-value language SCHEME. Furthermore PLOTKIN only considers the integers and the booleans as base types, whereas we need complex recursively defined types as base types (see section 3.2). We leave the problem of proving adequacy of our denotational semantics for a fragment of a call-by-value language suitable for formalizing our extension of NbE to future work.

2 A simply typed λ-calculus with constants

2.1 Types, terms, rewrite rules

We start from a given set of *ground types*. *Types* are inductively generated from ground types τ by $\rho \to \sigma$ and $\rho \times \sigma$. *Terms* are

x^ρ	typed variables,
c^ρ	constants,
$(\lambda x^\rho M^\sigma)^{\rho\to\sigma}$	abstractions,
$(M^{\rho\to\sigma} N^\rho)^\sigma$	applications,
$\langle M_0^\rho, M_1^\sigma \rangle^{\rho\times\sigma}$	pairing,
$\pi_0(M^{\rho\times\sigma})^\rho$, $\pi_1(M^{\rho\times\sigma})^\sigma$	projections.

Type indices will be omitted whenever they are inessential or clear from the context. Also, λx binds tighter than application and pairing; however, a dot after λx means that the scope extends as far as allowed by the parentheses. So $\lambda x M N$ means $(\lambda x M)N$, but $\lambda x.MN$ means $\lambda x(MN)$.

Ground types will always be denoted by τ. We sometimes write $M0$ for $\pi_0(M)$ and $M1$ for $\pi_1(M)$. Two terms M and N are called *α-equal* – written

$M =_\alpha N$ – if they are equal up to renaming of bound variables. Λ_ρ denotes the set of all terms of type ρ (α-equal terms are *not* identified). $M\boldsymbol{N}$ denotes $(\dots(MN_1)N_2\dots)N_n$, where some of the N_i's may be 0 or 1. By $\mathsf{FV}(M)$ we denote a list of variables occurring free in M. By $M_x[N]$ we mean substitution of every free occurrence of x in M by N, renaming bound variables if necessary. Similarly $M_{\boldsymbol{x}}[\boldsymbol{N}]$ denotes simultaneous substitution. $\lambda\boldsymbol{x}M$ abbreviates $\lambda x_1\dots\lambda x_n M$. If $M\boldsymbol{N}$ is of type σ, N_i of type ρ_i, then we call $\boldsymbol{\rho}\to\sigma$ a type information for M. Here $\boldsymbol{\rho}$ is a list of types, 0's or 1's indicating the left or right part of a product type. So e.g. a term M of type $\rho = (\tau\to\tau\to\tau)\times(\tau\to(\tau\times\tau))$ has $(0,\tau)\to(\tau\to\tau)$ or $(1,\tau,0)\to\tau$ as a type information. If there are no product types $\boldsymbol{\rho}\to\sigma$ simply abbreviates $(\rho_1\to(\rho_2\cdots\to(\rho_n\to\sigma)\dots))$.

For the constants c^ρ we assume that some rewrite rules of the form $c\boldsymbol{K}\longmapsto N$ are given, where $\mathsf{FV}(N)\subseteq\mathsf{FV}(\boldsymbol{K})$ and $c\boldsymbol{K}$, N have the same type (not necessarily a ground type). Moreover, for any type information $\rho_1,\dots\rho_n\to\tau$ for c (τ a ground type), we require that there is a fixed length $k\le n$ of arguments for the rewrite rules, i.e. $c\boldsymbol{M}\longmapsto N$ implies that $\boldsymbol{M}$ has length k, provided the projection markers in $\boldsymbol{M}$ and in $\rho_1,\dots\rho_k$ coincide. If no rewrite rule of the form $c\boldsymbol{M}\longmapsto N$ ($1\le$ length of $\boldsymbol{M}\le n$) applies, then this fixed length is stipulated to be n. We write $c^{\boldsymbol{\rho}\to\sigma}$ to indicate that we only consider c with argument lists $\boldsymbol{K}$ with these projection markers; the notation $c\boldsymbol{M}\boldsymbol{N}$ is used to indicate that $\boldsymbol{M}$ are the fixed arguments for the rewrite rules of c. In particular. if there is no rewrite rule for c, then $\boldsymbol{N}$ is empty and $c\boldsymbol{M}$ is of ground type.

For example, if c is of type $(\tau\to\tau\to\tau)\times(\tau\to\tau)$, then the rules $c0xx\longmapsto a$ and $c1\longmapsto b$ are admitted, and $c^{0,\tau,\tau\to\tau}$ indicates that we only consider argument lists of the form $0,x,y$.

2.2 Computation rules

Given a set of rewrite rules, we want to treat some rules - which we call *computation rules* - in a different, more efficient way. The idea is that a computation rule can be understood as a description of a computation in a suitable *semantical* model, provided the syntactic constructors correspond to semantic ones in the model, whereas the other rules describe *syntactic* transformations.

A constant c is called a *constructor* if there is no rule of the form $c\boldsymbol{K}\longmapsto N$. For instance in the examples of section 2.3 the constants 0, S and $\exists^+$ are constructors. *Constructor patterns* are special terms defined inductively as follows.

- Every variable is a constructor pattern.
- If c is a constructor and $P_1,\dots,P_n$ are constructor patterns or projection markers 0 or 1, such that $c\boldsymbol{P}$ is of ground type, then $c\boldsymbol{P}$ is a constructor pattern.

From the given set of rewrite rules we choose a subset COMP with the following properties.

- If $c\boldsymbol{P} \longmapsto Q \in \textsc{Comp}$, then $P_1, \ldots, P_n$ are constructor patterns or projection markers.
- The rules are left-linear, i.e. if $c\boldsymbol{P} \longmapsto Q \in \textsc{Comp}$, then every variable in $c\boldsymbol{P}$ occurs only once in $c\boldsymbol{P}$.
- The rules are non-overlapping, i.e. for different rules $c\boldsymbol{K} \longmapsto M$ and $c\boldsymbol{L} \longmapsto N$ in $\textsc{Comp}$ the left hand sides $c\boldsymbol{K}$ and $c\boldsymbol{L}$ are non-unifiable.

We write $c\boldsymbol{M} \longmapsto_{\mathsf{comp}} Q$ to indicate that the rule is in $\textsc{Comp}$. The set of constructors appearing in the constructor patterns is denoted by $\textsc{Constr}$. All other rules will be called (proper) rewrite rules, written $c\boldsymbol{M} \longmapsto_{\mathsf{rew}} K$.

In our reduction strategy below computation rules will always be applied first, and since they are non-overlapping, this part of the reduction is unique. However, since we allowed almost arbitrary rewrite rules, it may happen that in case no computation rule applies a term may be rewritten by different rules $\notin \textsc{Comp}$. In order to obtain a deterministic procedure we assume that for every constant $c^{\boldsymbol{\rho}\to\sigma}$ we are given a function sel_c computing from $\boldsymbol{M}$ either a rule $c\boldsymbol{K} \longmapsto_{\mathsf{rew}} N$, in which case $\boldsymbol{M}$ is an instance of $\boldsymbol{K}$, i.e. $\boldsymbol{M} = \boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}]$, or else the message "$\mathsf{no\text{-}match}$", in which case $\boldsymbol{M}$ doesn't match any rewrite rule, i.e. there is no rule $c\boldsymbol{K} \longmapsto_{\mathsf{rew}} N$ such that $\boldsymbol{M}$ is an instance of $\boldsymbol{K}$. Clearly sel_c should be compatible with α-equality, and should satisfy an obvious *uniformity* property, i.e. whenever $\boldsymbol{M}$ and $\boldsymbol{M}'$ are variants (i.e. can be obtained from each other by an invertible substitution), then $\mathsf{sel}_c(\boldsymbol{M}) = \mathsf{sel}_c(\boldsymbol{M}')$.

Often the rewrite rules will be left-linear (i.e. no variable occurs twice in the left hand side of a rule); then is is reasonable to require that every select function sel_c is *strongly uniform* in the sense that for all instances (with not necessarily distinct variables $\boldsymbol{z}$) we have $\mathsf{sel}_c(\boldsymbol{M}) = \mathsf{sel}_c(M_{\boldsymbol{x}}[\boldsymbol{z}])$.

2.3 Examples

(a) Usually we have the ground type ι of natural numbers available, with constructors 0^ι, $\mathsf{S}^{\iota\to\iota}$ and *recursion operators* $R_\rho^{\iota\to\rho\to(\iota\to\rho\to\rho)\to\rho}$. The rewrite rules for R are

$$\begin{aligned} R0 &\longmapsto \lambda yz.y, \\ R(\mathsf{S}x) &\longmapsto \lambda yz.zx(Rxyz). \end{aligned}$$

The reason for writing the rules in this way, and not in the more familiar form $R0yz \longmapsto y$, $R(\mathsf{S}x)yz \longmapsto zx(Rxyz)$, will become clear later (see example 16 in section 3.5). A simplified scheme of a similar form gives a cases construct.

$$\begin{aligned} \mathsf{if}\, 0 &\longmapsto \lambda yz.y, \\ \mathsf{if}\, (\mathsf{S}x) &\longmapsto \lambda yz.z. \end{aligned}$$

Moreover we can write down rules according to the usual recursive definitions of addition and multiplication, e.g.

$$\mathsf{mult}(\mathsf{S}x) \longmapsto \lambda z.\mathsf{add}\, z(\mathsf{mult}\, xz)$$

Simultaneous recursion may be treated as well, e.g.

$$\begin{aligned} \mathsf{odd}\,0 &\longmapsto \mathsf{S}0 & \mathsf{even}\,0 &\longmapsto 0,\\ \mathsf{odd}(\mathsf{S}x) &\longmapsto \mathsf{even}\,x & \mathsf{even}(\mathsf{S}x) &\longmapsto \mathsf{odd}\,x. \end{aligned}$$

All these rules are possible computation rules, whereas the next two are rules are not (since if and add are no constructors).

$$\mathsf{if}\,(\mathsf{if}\,xyz) \longmapsto \lambda uv.\mathsf{if}\,x(\mathsf{if}\,yuv)(\mathsf{if}\,zuv),$$

(a rewrite rule due to McCarthy [12]), or

$$\mathsf{mult}(\mathsf{add}\,xy) \longmapsto \lambda z.\mathsf{add}(\mathsf{mult}\,xz)(\mathsf{mult}\,yz).$$

(b) We can also deal with infinitely branching trees like the Brouwer ordinals of type $\mathcal{O}$. There are constructors $0^{\mathcal{O}}$ and $\mathrm{Sup}^{(\iota\to\mathcal{O})\to\iota}$, and recursion constants $\mathrm{Rec}_{\rho}^{\mathcal{O}\to\rho\to((\iota\to\mathcal{O})\to(\iota\to\rho)\to\rho)\to\rho}$. The rewrite rules for Rec are

$$\begin{aligned} \mathrm{Rec}\;0 &\longmapsto \lambda yz.y,\\ \mathrm{Rec}(\mathrm{Sup}\;x) &\longmapsto \lambda yz.zx(\lambda u\,\mathrm{Rec}(xu)yz). \end{aligned}$$

(c) It is well known that by the Curry-Howard correspondence natural deduction proofs can be written as λ-terms with formulas as types. To use normalization by evaluation for normalizing proofs we may also introduce a ground type ex with constructors and destructors

$$(\exists^+_{\rho_0,\rho_1})^{\rho_0\to\rho_1\to\mathsf{ex}} \quad \text{and} \quad (\exists^-_{\rho_0,\rho_1,\sigma})^{\mathsf{ex}\to(\rho_0\to\rho_1\to\sigma)\to\sigma};$$

these are called *existential constants*. The rewrite rule for $\exists^-$ is

$$\exists^-(\exists^+ x_0x_1) \longmapsto \lambda y.yx_0x_1.$$

The (constructive) existential quantifier can then be dealt with conveniently by means of axioms

$$\begin{aligned} &\exists^+ : \forall x(A \to \exists x A),\\ &\exists^- : \exists x A \to \forall x(A \to B) \to B \quad \text{with } x \notin \mathsf{FV}(B). \end{aligned}$$

If x has type ρ_0 and the formulas A and B are associated with the types ρ_1 and σ, respectively, the rewrite rule above is clear. It seems that the existential type ex could be replaced by $\rho_0\times\rho_1$ and the constants $\exists^+_{\rho_0,\rho_1}$ and $\exists^-_{\rho_0,\rho_1,\sigma}$ by the terms $\lambda x_0\lambda x_1\langle x_0,x_1\rangle$ and $\lambda z\lambda f(f\pi_0(z)\pi_1(z))$ respectively. However, the latter term does not correspond to a derivation in first order logic, since it is impossible to pass from an arbitrary derivation d (possibly with free assumptions) of $\exists xA$ to a term $\pi_0(d)$ and a derivation $\pi_1(d)$ of $A_x[\pi_0(d)]$.

One can easily formulate rules for *permutative conversions*, which permute an application of an $\exists$-elimination rule with other elimination rules, e.g.

$$\exists^-_{\rho_0,\rho_1,\sigma_0\to\sigma_1}p \longmapsto \lambda zv.\exists^-_{\rho_0,\rho_1,\sigma_1}p(\lambda xy.(zxyv)).$$

2.4 Normalizable terms and their normal forms

We inductively define a relation $M \longrightarrow Q$ for terms M, Q. The intended meaning of $M \longrightarrow Q$ is that M is normalizable with (long) normal form Q. However, it is necessary to split up $\longrightarrow$ into two relations: a "weak" one $\longrightarrow_w$ intended to unwrap the outer constructor form, followed by a "strong" one $\longrightarrow_s$, where we assume that it is applied to terms M irreducible w.r.t. $\longrightarrow_w$.

Looking at the form of a term we will embark on the following strategy:

- β-redexes $(\lambda x M)N$ and computation rules $c\boldsymbol{M}\boldsymbol{N}$ are reduced promptly, i.e. we use call-by-name here.
- If no rule applies to $c\boldsymbol{M}\boldsymbol{N}$ one first tries to find out whether $\boldsymbol{M}$ can be reduced to $\boldsymbol{P}$ such that $c\boldsymbol{P}$ matches a computation rule. This does not require reducing each M_i to normal form, it suffices to find out the outer pattern of M_i (let us call it for now "constructor normal form"). The reductions for doing so will be called "weak" and we write $\longrightarrow_w$ for them.
- If in $c\boldsymbol{M}\boldsymbol{N}$ all $\boldsymbol{M}$ are already in constructor normal form and no computation rule applies, then in a second step one reduces all $\boldsymbol{M}$ and $\boldsymbol{N}$ to normal form (if it exists) and tries to apply a proper rewrite rule, i.e. we use call-by-value at this point.

Let $\boldsymbol{M} \longrightarrow \boldsymbol{M}'$ abbreviate $M_1 \longrightarrow M_1', \ldots, M_n \longrightarrow M_n'$ and similarly for other relations, and $\longrightarrow_w^*$ be the reflexive and transitive closure of $\longrightarrow_w$.

Definition 1. Split.

$$\frac{M \longrightarrow_w^* N \quad N \longrightarrow_s Q}{M \longrightarrow Q}$$

Eta.

$$\frac{My \longrightarrow Q}{M^{\rho\to\sigma} \longrightarrow_s \lambda y Q} \quad \text{for } y \notin \mathsf{FV}(M) \qquad \frac{M0 \longrightarrow Q_0 \quad M1 \longrightarrow Q_1}{M^{\rho\times\sigma} \longrightarrow_s \langle Q_0, Q_1\rangle}$$

VarApp.

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M}'}{x\boldsymbol{M} \longrightarrow_s x\boldsymbol{M}'}, \quad \text{provided } x\boldsymbol{M} \text{ is of ground type.}$$

Beta.

$$(\lambda x M)N\boldsymbol{P} \longrightarrow_w M_x[N]\boldsymbol{P} \qquad \langle M_0, M_1\rangle i\boldsymbol{P} \longrightarrow_w M_i\boldsymbol{P} \quad \text{for } i \in \{0,1\}.$$

Comp.

$$c\boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N} \longrightarrow_w Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N} \quad \text{if } c\boldsymbol{P} \longmapsto_{\mathsf{comp}} Q.$$

For the next three rules assume that $c\boldsymbol{M}$ is not an instance of a computation rule.

Arg.

$$\frac{\boldsymbol{M} \longrightarrow_w^* \boldsymbol{M}'}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_w c\boldsymbol{M}'\boldsymbol{N}} \quad \text{with at least one } \longrightarrow_w\text{-reduction in } \boldsymbol{M} \longrightarrow_w^* \boldsymbol{M}'.$$

The final two rules have premises $\boldsymbol{M} \longrightarrow_s \boldsymbol{M}'$. Note that by lemma 2 below, $c\boldsymbol{M'}$ cannot be an instance of a computation rule, for then also $c\boldsymbol{M}$ would be one.

Rew.

$$\frac{\boldsymbol{M} \longrightarrow_s \boldsymbol{M}'}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_w Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}} \quad \text{if } \mathsf{sel}_c(\boldsymbol{M'}) = c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q \text{ and } \boldsymbol{M'} = \boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}].$$

PassApp.

$$\frac{\boldsymbol{M} \longrightarrow_s \boldsymbol{M}' \quad \boldsymbol{N} \longrightarrow \boldsymbol{N}'}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_s c\boldsymbol{M}'\boldsymbol{N}'} \quad \text{if } \mathsf{sel}_c(\boldsymbol{M'}) = \mathsf{no\text{-}match} \text{ and } c\boldsymbol{M}\boldsymbol{N} \text{ of ground type.}$$

In case the constant c in the rules Arg and PassApp is a constructor, $\boldsymbol{N}$ is required to be empty.

For readability we will often write Rew in the following form, assuming that $c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q$ is the selected rule.

Rew.

$$\frac{\boldsymbol{M} \longrightarrow_s \boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}]}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_w Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}} \quad \text{if } c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q.$$

For the definition above to make sense we prove the following.

Lemma 2. *If $M \longrightarrow_s M'$ and M' is an instance of a constructor pattern P, then also M is an instance of P.*

Proof. By induction on P. If P is a variable the claim is trivial, so let $P = c\boldsymbol{P}$. Then $M' = c\boldsymbol{K}'$ and $\boldsymbol{K}'$ is an instance of $\boldsymbol{P}$. Moreover, the only possibility to infer $M \longrightarrow_s M' = c\boldsymbol{K}'$ is by PassApp. Thus $M = c\boldsymbol{K}$, $\boldsymbol{K} \longrightarrow_s \boldsymbol{K}'$ and by induction hypothesis (IH) $\boldsymbol{K}$ is an instance of $\boldsymbol{P}$. Since $\boldsymbol{P}$ is linear we eventually get that $c\boldsymbol{K}$ is an instance of $c\boldsymbol{P}$. □

Definition 3. The set Lnf of terms in long normal form is defined as follows. $\lambda x M$, $\langle M, N\rangle$, $(x\boldsymbol{M})^\tau$ and $(c\boldsymbol{M}\boldsymbol{N})^\tau$ are in Lnf if $M, N, \boldsymbol{M}, \boldsymbol{N}$ are, provided that $c\boldsymbol{M}$ is not an instance of any computation or rewrite rule.

For example, the *η-expansion* $\mathsf{exp}(x)$ of a variable x is in long normal form; it is defined using induction on types by (e.g. for pure $\to$-types) $\mathsf{exp}(x^\tau) = x^\tau$, $\mathsf{exp}(x^{\boldsymbol{\rho}\to\tau}) = \lambda \boldsymbol{y}^{\boldsymbol{\rho}}.x\,\mathsf{exp}(\boldsymbol{y}^{\boldsymbol{\rho}})$.

Lemma 4. *If $M \longrightarrow Q$ or $M \longrightarrow_s Q$, then Q is in long normal form.*

Proof. By simultaneous induction on $M \longrightarrow Q$ and $M \longrightarrow_s Q$. The only interesting case is PassApp, where we have to show that $c\boldsymbol{M}'$ is not an instance of a computation rule. But if $c\boldsymbol{M}'$ would be such an instance, by the previous lemma $c\boldsymbol{M}$ would also be, contradicting the assumption. □

Furthermore it can be shown easily that if $M \longrightarrow Q$, $M \longrightarrow_w Q$ or $M \longrightarrow_s Q$, then M reduces to Q in the usual sense w.r.t. β-reduction, η-expansion and the computation and rewrite rules for the constants. However, the converse is not true in general. For a counterexample, consider the non-terminating rewrite rules $\mathsf{mult}\, x^{\iota} 0 \longmapsto_{\mathsf{rew}} 0$ and $\bot^{\iota} \longmapsto_{\mathsf{rew}} \bot$. Then 0 is a normal form of $\mathsf{mult}\bot 0$, but we cannot have $\mathsf{mult}\bot 0 \longrightarrow Q$ for any Q. To see this, note that we cannot have $\bot \longrightarrow_s N$ for any N (since $\bot \longmapsto_{\mathsf{rew}} \bot$), hence we also cannot have $\mathsf{mult}\bot 0 \longrightarrow_s Q$ for any Q. Since $\bot, 0$ are $\longrightarrow_w^*$-reducible only to themselves, the claim follows. – But under the hypothesis that M is *strongly* normalizable the converse is true.

Lemma 5. *If M is strongly normalizable w.r.t. these reductions (i.e. every reduction sequence terminates), then $M \longrightarrow Q$ for some Q.*

Proof. For simplicity we consider pure $\to$-types only; the extension to product types is immediate. We will prove the claim by induction on h_M and side induction on $\mathsf{ht}(M)$, where h_M denotes the height of the reduction tree for M and $\mathsf{ht}(M)$ is the height of M. Note that if $M \longrightarrow_w Q$ then M reduces to Q in at least one step, hence $h_M > h_Q$.

Case $\lambda y M$. We have $(\lambda y M)y \longrightarrow_w M \longrightarrow Q$ by Beta and the side induction hypothesis (SIH), hence $\lambda y M \longrightarrow \lambda y Q$ by Eta.

Case M has a type $\rho \to \sigma$, but is not an abstraction. Then M η-expands to $\lambda y.My$ where y is a new variable of type ρ, hence $h_M > h_{\lambda y.My} \geq h_{My}$. Therefore $My \longrightarrow Q$ by IH. Hence $M \longrightarrow \lambda y Q$ by Eta.

It remains to consider terms of ground type.

Case $x\boldsymbol{M}$. Obvious, using the SIH and rule VarApp.

Case $(\lambda x M)N\boldsymbol{P}$. Then $(\lambda x M)N\boldsymbol{P} \longrightarrow_w M_x[N]\boldsymbol{P} \longrightarrow Q$ by Beta and the IH.

Case $c\boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}$ with $c\boldsymbol{P} \longmapsto_{\mathsf{comp}} Q$. Then $c\boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N} \longrightarrow_w Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N} \longrightarrow Q_1$ by Comp and the IH.

Case $c\boldsymbol{M}\boldsymbol{N}$ with $c\boldsymbol{M}$ not an instance of a computation rule. By SIH $\boldsymbol{M} \longrightarrow \boldsymbol{M}'$. If at least one M_i is $\longrightarrow_w$-reduced, the claim follows from the IH and Arg. Otherwise we have $\boldsymbol{M} \longrightarrow_s \boldsymbol{M}'$. Now if $\mathsf{sel}_c(\boldsymbol{M}') = c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q$ and $\boldsymbol{M}' = \boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}]$, the claim follows from the IH for $Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}$. If however $\mathsf{sel}_c(\boldsymbol{M}') = \mathsf{no\text{-}match}$, then proceed as in case $x\boldsymbol{M}$, using PassApp instead of VarApp. □

Moreover, the relation $M \longrightarrow Q$ clearly is not closed under substitution. However, it is closed under substitution of variables, provided the result is a variant of M.

Lemma 6. *Let $\to \in \{\longrightarrow, \longrightarrow_w, \longrightarrow_s\}$. If $M \to Q$, then $M_{\boldsymbol{x}}[\boldsymbol{z}] \to Q_{\boldsymbol{x}}[\boldsymbol{z}]$ with a derivation of the same height, provided $\boldsymbol{z}$ are distinct variables $\notin \mathsf{FV}(M)$*[1]*.*

Proof. We use induction on the height of the derivation of $M \to Q$. Clearly we may assume $\boldsymbol{x} \in \mathsf{FV}(M)$.

Case Eta.

$$\frac{My \longrightarrow Q}{M^{\rho\to\sigma} \longrightarrow_s \lambda y Q} \quad \text{for } y \notin \mathsf{FV}(M).$$

[1] If we assume strong uniformity of all sel_c-functions (as we implicitly did in [4]), then the proviso is not necessary.

Recall

$$(\lambda y Q)_{\boldsymbol{x}}[\boldsymbol{z}] := \begin{cases} \lambda u Q_{\boldsymbol{x},y_i}[\boldsymbol{z},u] & \text{if } y = z_i \text{ for some } i \text{ with } x_i \in \mathsf{FV}(Q) \\ \lambda y Q_{\boldsymbol{x}}[\boldsymbol{z}] & \text{otherwise} \end{cases}$$

with a new variable u.

Subcase 1. $y = z_i$ for some i with $x_i \in \mathsf{FV}(Q)$. By IH $M_{\boldsymbol{x}}[\boldsymbol{z}]u \longrightarrow Q_{\boldsymbol{x},y}[\boldsymbol{z},u]$, hence $M_{\boldsymbol{x}}[\boldsymbol{z}] \longrightarrow_s \lambda u Q_{\boldsymbol{x},y}[\boldsymbol{z},u]$ by Eta.

Subcase 2. y not in $\boldsymbol{z}$. Because of $\boldsymbol{x} \in \mathsf{FV}(M)$ we have y not in $\boldsymbol{x}$. Hence $M_{\boldsymbol{x}}[\boldsymbol{z}]y = (My)_{\boldsymbol{x}}[\boldsymbol{z}] \longrightarrow Q_{\boldsymbol{x}}[\boldsymbol{z}]$ by IH and the claim follows by Eta.

Subcase 3. $y = z_i$ for some i with $x_i \notin \mathsf{FV}(Q)$. Let $\hat{\boldsymbol{z}}$ be $\boldsymbol{z}$ without z_i and $\hat{\boldsymbol{x}}$ be $\boldsymbol{x}$ without x_i. Then $Q_{\boldsymbol{x}}[\boldsymbol{z}] = Q_{\hat{\boldsymbol{x}}}[\hat{\boldsymbol{z}}]$, hence $\lambda y Q_{\hat{\boldsymbol{x}}}[\hat{\boldsymbol{z}}] = (\lambda y Q)_{\hat{\boldsymbol{x}}}[\hat{\boldsymbol{z}}] = (\lambda y Q)_{\boldsymbol{x}}[\boldsymbol{z}]$ and the claim follows as in subcase 2.

Case Rew.

$$\frac{\boldsymbol{M} \longrightarrow_s \boldsymbol{M}'}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_w Q_{\boldsymbol{y}}[\boldsymbol{L}]\boldsymbol{N}} \quad \text{if } \mathsf{sel}_c(\boldsymbol{M}') = c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q \text{ and } \boldsymbol{M}' = \boldsymbol{K}_{\boldsymbol{y}}[\boldsymbol{L}].$$

Let $\boldsymbol{z}$ be distinct variables $\notin \mathsf{FV}(c\boldsymbol{M}\boldsymbol{N})$. We want to derive $(c\boldsymbol{M}\boldsymbol{N})_{\boldsymbol{x}}[\boldsymbol{z}] \longrightarrow_w (Q_{\boldsymbol{y}}[\boldsymbol{L}]\boldsymbol{N})_{\boldsymbol{x}}[\boldsymbol{z}]$ by Rew again. Clearly we may assume $\boldsymbol{x}, \boldsymbol{z} \notin \mathsf{FV}(\boldsymbol{K}, Q)$. By IH $\boldsymbol{M}_{\boldsymbol{x}}[\boldsymbol{z}] \longrightarrow_s \boldsymbol{M}'_{\boldsymbol{x}}[\boldsymbol{z}]$ and $\boldsymbol{z} \notin \mathsf{FV}(\boldsymbol{M}')$ (since $\boldsymbol{M} \longrightarrow_s \boldsymbol{M}'$ implies that $\boldsymbol{M}'$ has no more free variables than $\boldsymbol{M}$). Now $\mathsf{sel}_c(\boldsymbol{M}'_{\boldsymbol{x}}[\boldsymbol{z}]) = \mathsf{sel}_c(\boldsymbol{M}') = c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q$ by uniformity of sel_c, and $\boldsymbol{M}'_{\boldsymbol{x}}[\boldsymbol{z}] = \boldsymbol{K}_{\boldsymbol{y}}[\boldsymbol{L}_{\boldsymbol{x}}[\boldsymbol{z}]]$. An application of Rew yields $(c\boldsymbol{M}\boldsymbol{N})_{\boldsymbol{x}}[\boldsymbol{z}] = c\boldsymbol{M}_{\boldsymbol{x}}[\boldsymbol{z}]\boldsymbol{N}_{\boldsymbol{x}}[\boldsymbol{z}] \longrightarrow_w Q_{\boldsymbol{y}}[\boldsymbol{L}_{\boldsymbol{x}}[\boldsymbol{z}]]\boldsymbol{N}_{\boldsymbol{x}}[\boldsymbol{z}] = (Q_{\boldsymbol{y}}[\boldsymbol{L}]\boldsymbol{N})_{\boldsymbol{x}}[\boldsymbol{z}]$.

The simplifications in case we assume strongly uniformity of all sel_c-functions are obvious. □

Example 7. In the examples in section 2.3, many of the (proper) rewrite rules have been written as "higher type rules", i.e. in the form $c\boldsymbol{M} \longmapsto \lambda x N$ rather than $c\boldsymbol{M}x \longmapsto N$. This is preferable from a semantical point of view, because the latter form may cause unnecessary calculations (cf. the definition of $\mathcal{I}(c)$ in section 3.5). Note also that in the presence of non-terminating rewrite rules both versions can lead to different sets of normalizing terms. To see that, assume there is a term M that has no normal form, so no N exists such that $M \longrightarrow_s N$. Then for a proper rewrite rule $cy \longmapsto d$ the term cM has no normal form, but for $c \longmapsto \lambda y d$ it has, namely d.

Also for terminating but non-confluent rewrite rules both versions can lead to different normal forms. Here is an example, with all rules considered as proper rewrite rules.

$$\begin{array}{rll} d2x \longmapsto 2 & & d \text{ of type } \tau \to \tau \to \tau \\ dx3 \longmapsto 3 = \mathsf{sel}_d(2,3) & & \\ cy \longmapsto y3 & & c \text{ of type } (\tau \to \tau) \to \tau \end{array}$$

Then

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{2 \longrightarrow_s 2 \qquad x \longrightarrow_s x}{d2x \longrightarrow_w 2 \longrightarrow_s 2}\,\textsc{Rew}}{d2x \longrightarrow 2}\,\textsc{Split}}{d2 \longrightarrow_s \lambda x2}\,\textsc{Eta}}{c(d2) \longrightarrow_w (\lambda x2)3 \longrightarrow_w 2 \longrightarrow_s 2}\,\textsc{Rew}}{c(d2) \longrightarrow 2}\,\textsc{Split}$$

If, however, the last rule is replaced by

$$c \longmapsto \lambda y.y3,$$

we obtain $c(d2) \longrightarrow_w (\lambda y.y3)(d2)$ by Rew, hence

$$\dfrac{c(d2) \longrightarrow_w (\lambda y.y3)(d2) \longrightarrow_w d23 \longrightarrow_w 3 \longrightarrow_s 3}{c(d2) \longrightarrow 3}\,\textsc{Split}$$

Example 8. Formally it is possible to add a fixpoint operator Y with rewrite rule $\mathsf{Y} \longmapsto \lambda x.x(\mathsf{Y}x)$ or $\mathsf{Y}x \longmapsto x(\mathsf{Y}x)$. But if one tries to define e.g. addition add by a fixpoint operator, $\mathsf{add}\,xy$ would have no normal form:

$$\mathsf{add} := \lambda x.\mathsf{Y}M \text{ where } M := \lambda zy.\mathsf{if}\ yx(\mathsf{S}(z(\mathsf{P}y))).$$

If $\mathsf{add}\,xy$ would have a normal form with respect to our rules, there must be a term N such that

$$(\lambda x.\mathsf{Y}M)xy \longrightarrow N.$$

To derive this relation it is necessary to show $\mathsf{Y}My \longrightarrow N$, and by the rewrite rule for the fixpoint operator we would have to show $M(\mathsf{Y}M)y \longrightarrow N$, hence

$$\mathsf{if}\ yx(\mathsf{S}(\mathsf{Y}M(\mathsf{P}y))) \longrightarrow N.$$

But by the rule Arg we now need to have a normal form for $\mathsf{Y}M(\mathsf{P}y)$, and thus for $\mathsf{Y}M(\mathsf{P}(\mathsf{P}y))$, and so on. – However, the term $\mathsf{add}\,MN$ with numerals M, N reduces to a numeral.

2.5 Term families

Since normalization by evaluation needs to create bound variables when "reifying" abstract objects of higher type, it is useful to follow de Bruijn's [9] style of representing bound variables in terms. This is done here – as in [5, 10] – by means of *term families*. A term family is a parametrized version of a given term M. The idea is that the term family of M at index k reproduces M with bound variables renamed starting at k. For example, for

$$M := \lambda u\lambda v.c(\lambda x.vx)(\lambda y\lambda z.zu)$$

the associated term family M^∞ at index 3 yields

$$M^\infty(3) := \lambda x_3\lambda x_4.c(\lambda x_5.x_4x_5)(\lambda x_5\lambda x_6.x_6x_3).$$

We denote terms by $M, N, K, \dots$, and term families by $r, s, t, \dots$.

To every term M^ρ we assign a term family $M^\infty\colon \mathbb{N} \to \Lambda_\rho$ by

$$\begin{aligned} x^\infty(k) &:= x, \\ c^\infty(k) &:= c, \\ (\lambda y M)^\infty(k) &:= \lambda x_k (M_y[x_k]^\infty(k+1)), & \langle M_0, M_1\rangle^\infty(k) &:= \langle M_0^\infty(k), M_1^\infty(k)\rangle, \\ (MN)^\infty(k) &:= M^\infty(k) N^\infty(k), & \pi_i(M)^\infty(k) &:= \pi_i(M^\infty(k)). \end{aligned}$$

Application of a term family $r\colon \mathbb{N} \to \Lambda_{\rho\to\sigma}$ to a term family $s\colon \mathbb{N} \to \Lambda_\rho$ is the family $rs\colon \mathbb{N} \to \Lambda_\sigma$ defined by $(rs)(k) := r(k)s(k)$, and similarly for pairing $\langle r_0, r_1\rangle(k) := \langle r_0(k), r_1(k)\rangle$ and projections $\pi_i(r)(k) := \pi_i(r(k))$. Hence e.g. $(MN)^\infty = M^\infty N^\infty$.

We let $k > \mathsf{FV}(M)$ mean that k is greater than all i such that $x_i^\rho \in \mathsf{FV}(M)$ for some type ρ.

Lemma 9. *a. If $M =_\alpha N$, then $M^\infty = N^\infty$.*

b. If $k > \mathsf{FV}(M)$, then $M^\infty(k) =_\alpha M$.

Proof. a. Induction on the height $\mathsf{ht}(M)$ of M. Only the case where M and N are abstractions is critical. So assume $\lambda y^\rho M =_\alpha \lambda z^\rho N$. Then $M_y[P] =_\alpha N_z[P]$ for all terms P^ρ. In particular $M_y[x_k] =_\alpha N_z[x_k]$ for arbitrary $k \in \mathbb{N}$. Hence $M_y[x_k]^\infty(k+1) = N_z[x_k]^\infty(k+1)$, by IH. Therefore

$$(\lambda y M)^\infty(k) = \lambda x_k(M_y[x_k]^\infty(k+1)) = \lambda x_k(N_z[x_k]^\infty(k+1)) = (\lambda z N)^\infty(k).$$

b. Induction on $\mathsf{ht}(M)$. We only consider the case $\lambda y M$. The assumption $k > \mathsf{FV}(\lambda y M)$ implies $x_k \notin \mathsf{FV}(\lambda y M)$ and hence $\lambda y M =_\alpha \lambda x_k(M_y[x_k])$. Furthermore $k+1 > \mathsf{FV}(M_y[x_k])$, and hence $M_y[x_k]^\infty(k+1) =_\alpha M_y[x_k]$, by IH. Therefore

$$(\lambda y M)^\infty(k) = \lambda x_k(M_y[x_k]^\infty(k+1)) =_\alpha \lambda x_k(M_y[x_k]) =_\alpha \lambda y M.$$

∎

Let $\mathsf{ext}(r) := r(k)$, where k is the least number greater than all i such that some variable of the form x_i^ρ occurs (free or bound) in $r(0)$.

Lemma 10. *$\mathsf{ext}(M^\infty) =_\alpha M$.*

Proof. $\mathsf{ext}(M^\infty) = M^\infty(k)$ for the least $k > i$ for all i such that x_i^ρ occurs (free or bound) in $M^\infty(0)$, hence $k > \mathsf{FV}(M)$. Now use part b of the lemma above. ∎

For our interpretation of types in section 3.2 we will also have to consider partial term families $r\colon \mathbb{N} \rightharpoonup \Lambda_\rho$. We extend application of term families, rs, as well as the operation ext to partial term families in the obvious way following the principle that all syntactic operations are strict, i.e. undefined whenever one argument is undefined.

3 Normalization by evaluation

3.1 Domain theoretic semantics of simply typed λ-calculi

In this section, we shall discuss the domain theoretic semantics of simply typed lambda calculi in general. Although the constructions below are standard (see e.g. the books of Lambek/Scott [11] or Crole [7]), we discuss them in some detail in order to make the paper accessible also for readers not familiar with this subject. Most constructions make sense in an arbitrary cartesian closed category (ccc). However we will confine ourselves to the domain semantics and will only occasionally comment on the categorical aspects.

It is well-known that Scott-domains with continuous functions form a cartesian closed category Dom. The product $D \times E$ is the set-theoretic product with component-wise ordering. The exponential $[D \to E]$ is the continuous function space with pointwise ordering. The terminal object is the one point space $\mathbf{1} := \{\bot\}$ (there is no initial object and there are no coproducts). In order to cope with the categorical interpretation, we will identify an element x of a domain D with the mapping from $\mathbf{1}$ to D with value x.

Besides the cartesian closedness, we also use the fact that Dom is closed under infinite products and that there is a fixed point operator $\mathrm{Fix}\colon (D \to D) \to D$ assigning to every continuous function $f\colon D \to D$ its least fixed point $\mathrm{Fix}(f) \in D$. Furthermore we will use that partial families of terms form a domain and some basic operations on terms and term families are continuous and hence exist as morphisms in the category. Any other ccc with these properties would do as well.

Notation. Elements of a product domain $D_1 \times \cdots \times D_n$ are written $[a_1, \ldots, a_n]$. If $f \in [D_1 \to [D_2 \to \ldots [D_n \to E] \ldots]]$ and $a_i \in D_i$, then $f(a_1, \ldots, a_n)$ or $f(\boldsymbol{a})$ stands for $f(a_1) \ldots (a_n)$.

An *interpretation* for a given system of ground types is a mapping $\mathcal{I}$ assigning to every ground type τ a domain $\mathcal{I}(\tau)$. Given such an interpretation we define domains $[\![\rho]\!]^{\mathcal{I}}$ for every type ρ by

$$[\![\tau]\!]^{\mathcal{I}} := \mathcal{I}(\tau), \quad [\![\rho \to \sigma]\!]^{\mathcal{I}} := [[\![\rho]\!]^{\mathcal{I}} \to [\![\sigma]\!]^{\mathcal{I}}], \quad [\![\rho \times \sigma]\!]^{\mathcal{I}} := [\![\rho]\!]^{\mathcal{I}} \times [\![\sigma]\!]^{\mathcal{I}}.$$

We write $[\![\rho_1, \ldots, \rho_n]\!]^{\mathcal{I}} := [\![\rho_1 \times \cdots \times \rho_n]\!]^{\mathcal{I}} = [\![\rho_1]\!]^{\mathcal{I}} \times \cdots \times [\![\rho_n]\!]^{\mathcal{I}} =: [\![\boldsymbol{\rho}]\!]^{\mathcal{I}}$. An *interpretation of a typed lambda calculus* (specified by a set of ground types and a set of constants) is a mapping $\mathcal{I}$ assigning to every ground type τ a domain $\mathcal{I}(\tau)$ (hence $\mathcal{I}$ is an interpretation of ground types), and assigning to every constant c^ρ a value $\mathcal{I}(c) \in [\![\rho]\!]^{\mathcal{I}}$ (i.e. a morphism from $\mathbf{1}$ to $[\![\rho]\!]^{\mathcal{I}}$).

In order to extend such an interpretation to all terms we use the following continuous functions, i.e. morphisms (in the sequel a continuous function will be called morphism if its role as a morphism in the ccc Dom is to be emphasized).

$$!_D\colon D \to \mathbf{1}, \quad !_D(d) := \bot$$
$$\pi_i\colon D_1 \times \cdots \times D_n \to D_i, \quad \pi_i([\boldsymbol{a}]) := a_i,$$
$$\mathsf{curry}\colon [D \times E \to F] \to [D \to [E \to F]], \quad \mathsf{curry}(f, a, b) := f([a, b]),$$

$$\mathsf{eval}\colon [D \to E] \times D \to E, \quad \mathsf{eval}([f, a]) := f(a).$$

Furthermore we use the fact that morphisms are closed under composition $\circ$ and (since DOM is a ccc) under pairing $\langle ., . \rangle$, where for $f\colon D \to E$ and $g\colon D \to F$ the function $\langle f, g\rangle\colon D \to E \times F$ maps a to $[f(a), g(a)]$. For every type ρ and every list of distinct variables $\boldsymbol{x}^{\boldsymbol{\rho}} = x_1^{\rho_1}, \dots, x_n^{\rho_n}$ we let $\Lambda_\rho(\boldsymbol{x})$ denote the set of terms of type ρ with free variables among $\{\boldsymbol{x}\}$. Let $\mathcal{I}$ be an interpretation. Then for every $M \in \Lambda_\rho(\boldsymbol{x}^{\boldsymbol{\rho}})$ we define a morphism $[\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}\colon [\![\boldsymbol{\rho}]\!] \to [\![\rho]\!]$ by

$$\begin{aligned}
[\![c]\!]^{\mathcal{I}}_{\boldsymbol{x}} &:= \mathcal{I}(c) \circ !_{[\![\boldsymbol{\rho}]\!]}, \\
[\![x_i]\!]^{\mathcal{I}}_{\boldsymbol{x}} &:= \pi_i, \\
[\![\lambda x M]\!]^{\mathcal{I}}_{\boldsymbol{x}} &:= \mathsf{curry}([\![M]\!]^{\mathcal{I}}_{\boldsymbol{x},x}), \\
[\![MN]\!]^{\mathcal{I}}_{\boldsymbol{x}} &:= \mathsf{eval} \circ [[\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}, [\![N]\!]^{\mathcal{I}}_{\boldsymbol{x}}], \\
[\![\langle M, N\rangle]\!]^{\mathcal{I}}_{\boldsymbol{x}} &:= [[\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}, [\![N]\!]^{\mathcal{I}}_{\boldsymbol{x}}], \\
[\![\pi_i(M)]\!]^{\mathcal{I}}_{\boldsymbol{x}} &:= \pi_i \circ [\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}.
\end{aligned}$$

This definition works in any ccc. For our purposes it will be more convenient to evaluate a term in a global environment and not in a local context. Let

$$\mathrm{ENV} := \prod_{x^\sigma \in \mathrm{VAR}} [\![\sigma]\!]^{\mathcal{I}} \in \mathrm{DOM}.$$

For every term $M \in \Lambda_\rho(x_1, \dots, x_n)$ we define a continuous function

$$[\![M]\!]^{\mathcal{I}}\colon \mathrm{ENV} \to [\![\rho]\!]^{\mathcal{I}}, \quad [\![M]\!]^{\mathcal{I}}_\xi := [\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}([\xi(x_1), \dots, \xi(x_n)]).$$

Formally this definition depends on a particular choice of the list of variables $x_1, \dots, x_n$. However, because of the well-known coincidence property in fact it does not.

From this we easily get the familiar equations

$$\begin{aligned}
[\![c]\!]^{\mathcal{I}}_\xi &= \mathcal{I}(c), \\
[\![x]\!]^{\mathcal{I}}_\xi &= \xi(x), \\
[\![\lambda x M]\!]^{\mathcal{I}}_\xi(a) &= [\![M]\!]^{\mathcal{I}}_{\xi[x \mapsto a]}, \\
[\![MN]\!]^{\mathcal{I}}_\xi &= [\![M]\!]^{\mathcal{I}}_\xi([\![N]\!]^{\mathcal{I}}_\xi), \\
[\![\langle M, N\rangle]\!]^{\mathcal{I}}_\xi &= [[\![M]\!]^{\mathcal{I}}_\xi, [\![N]\!]^{\mathcal{I}}_\xi], \\
[\![\pi_i(M)]\!]^{\mathcal{I}}_\xi &= \pi_i([\![M]\!]^{\mathcal{I}}_\xi).
\end{aligned}$$

In many cases the interpretation $\mathcal{I}$ of the constants will have to be defined recursively, by e.g. referring to $[\![M]\!]^{\mathcal{I}}$ for several terms M. This causes no problem, since the functionals $[\![M^\rho]\!]^{\mathcal{I}}\colon \mathrm{ENV} \to [\![\rho]\!]$ depend continuously on $\mathcal{I}$, where $\mathcal{I}$ is to be considered as an element of the infinite product $\Pi_{c^\rho}[\![\rho]\!]$. This can be seen as follows. Looking at their definitions we see that the functions

$[\mathcal{I}, \boldsymbol{a}] \mapsto [\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}(\boldsymbol{a})$ are built by composition from the continuous functions

$$\begin{aligned}
&\pi_{c^\sigma} \colon \Pi_{c^\rho}[\![\rho]\!] \to [\![\sigma]\!], \quad \pi_{c^\sigma}(\mathcal{I}) := \mathcal{I}(c),\\
&\cdot \circ \cdot \colon [E \to F] \times [D \to E] \to [D \to F],\\
&\langle \cdot, \cdot \rangle \colon [D \to E] \times [D \to F] \to [D \to E \times F],
\end{aligned}$$

as well as the functions $!_D, \pi_i, \mathsf{curry}$ and eval listed above. Hence $[\mathcal{I}, \boldsymbol{a}] \mapsto [\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}(\boldsymbol{a})$ is continuous. But then also $[\mathcal{I}, \xi] \mapsto [\![M]\!]^{\mathcal{I}}_{\xi}$ is continuous, since $[\![M]\!]^{\mathcal{I}}_{\xi} = [\![M]\!]^{\mathcal{I}}_{\boldsymbol{x}}([\pi_{x_1}(\xi), \dots, \pi_{x_n}(\xi)])$, where $\pi_{x^\rho} \colon \text{Env} \to [\![\rho]\!]$, $\pi_x(\xi) := \xi(x)$.

Hence the value $\mathcal{I}(c)$ may be defined as a least fixed point of a continuous function on the domain $\Pi_{c^\rho}[\![\rho]\!]$. – In the sequel we will omit the superscript $\mathcal{I}$ when it is clear from the context.

The following facts hold in any ccc.

Lemma 11.

$$\begin{aligned}
[\![M_{\boldsymbol{x}}[\boldsymbol{N}]]\!]_\xi &= [\![M]\!]_{\xi[\boldsymbol{x} \mapsto [\![\boldsymbol{N}]\!]_\xi]} && \textit{(substitution lemma)}\\
[\![(\lambda x M) N]\!]_\xi &= [\![M_x[N]]\!]_\xi && \textit{(beta 1)}\\
[\![\pi_i(\langle M_0, M_1 \rangle)]\!]_\xi &= [\![M_i]\!]_\xi && \textit{(beta 2)}\\
[\![M]\!]_\xi &= [\![\lambda y (M y)]\!]_\xi \quad (y^\rho \notin \mathsf{FV}(M^{\rho \to \sigma})) && \textit{(eta 1)}\\
[\![M]\!]_\xi &= [\![\langle \pi_0(M), \pi_1(M) \rangle]\!]_\xi && \textit{(eta 2)}
\end{aligned}$$

Lemma 12. *If $[\![P]\!]_\xi = [\![Q]\!]_\xi$ for all environments ξ, and M is transformed into N by replacing an occurrence of P in M by Q, then $[\![M]\!]_\xi = [\![N]\!]_\xi$ for all environments ξ.*

Proof. Induction on M. □

Lemma 13. *If M reduces to N by β-reduction or η-expansion, then $[\![M]\!]_\xi = [\![N]\!]_\xi$.* □

3.2 Interpretation of the types

We now consider a special model, whose ground type objects contain syntactic material. We let $\mathbb{N} \rightharpoonup \Lambda_\rho$ denote the set of partial term families, i.e. partial functions from the integers to the set of terms of type ρ. $\mathbb{N} \rightharpoonup \Lambda_\rho$ partially ordered by inclusion of graphs is a domain. We will interpret the ground types in such a way that we have functions

$$\downarrow_\tau \colon \mathcal{I}(\tau) \to (\mathbb{N} \rightharpoonup \Lambda_\tau) \qquad \text{and} \qquad \uparrow_\tau \colon (\mathbb{N} \rightharpoonup \Lambda_\tau) \to \mathcal{I}(\tau)$$

satisfying

$$\downarrow_\tau(\uparrow_\tau(r)) = r. \tag{1}$$

This shows that there is an embedding of the term families $\mathbb{N} \rightharpoonup \Lambda_\tau$ into $\mathcal{I}(\tau)$.

Recall that $\textsc{Constr}$ is the set of all constructors used in the computation rules. We define[2] the interpretation of the ground types in a way that all syntactic constructors $c \in \textsc{Constr}$ have semantical counterparts.

$$\mathcal{I}(\tau) = \sum([\![\boldsymbol{\rho}]\!]^{\mathcal{I}} \mid c^{\boldsymbol{\rho}\to\tau} \in \textsc{Constr}) + (\mathbb{N} \rightharpoonup \Lambda_\tau).$$

$\sum$ and $+$ denote the domain-theoretic separated[3] sum and

$$[\![\emptyset]\!] := \mathbf{1} := \{\bot\}, \quad [\![\boldsymbol{\rho}\sigma]\!] := [\![\boldsymbol{\rho}]\!] \times [\![\sigma]\!], \quad [\![\boldsymbol{\rho} i]\!] := [\![\boldsymbol{\rho}]\!] \text{ for } i \in \{0,1\}.$$

So for a given ground type τ, its constructors c together with term families $\mathbb{N} \rightharpoonup \Lambda_\tau$ freely generate the interpretation of τ, i.e. there are injections

$$\begin{aligned} \mathsf{in}_c^{\boldsymbol{\rho}} &: [\![\boldsymbol{\rho}]\!] \to [\![\tau]\!] && \text{for every } c^{\boldsymbol{\rho}\to\tau} \in \textsc{Constr} \\ \mathsf{fam}_\tau &: (\mathbb{N} \rightharpoonup \Lambda_\tau) \to [\![\tau]\!] \end{aligned}$$

such that every $a \in [\![\tau]\!]$ is either $\bot$, or else can be written uniquely as $a = \mathsf{in}_c^{\boldsymbol{\rho}}(\boldsymbol{b})$ or $a = \mathsf{fam}_\tau(r)$. For example (cf. section 2.3) we have

$$\begin{aligned} \mathcal{I}(\iota) &= \mathbf{1} + \mathcal{I}(\iota) + (\mathbb{N} \rightharpoonup \Lambda_\iota), \\ \mathcal{I}(\mathcal{O}) &= \mathbf{1} + [\mathcal{I}(\iota) \to \mathcal{I}(\mathcal{O})] + (\mathbb{N} \rightharpoonup \Lambda_{\mathcal{O}}), \\ \mathcal{I}(\mathsf{ex}) &= \sum([\![\rho]\!] \times [\![\sigma]\!] \mid \rho, \sigma \text{ types}) + (\mathbb{N} \rightharpoonup \Lambda_{\mathsf{ex}}). \end{aligned}$$

3.3 Reification and reflection

The continuous functions

$$\downarrow_\rho : [\![\rho]\!] \to (\mathbb{N} \rightharpoonup \Lambda_\rho) \quad (\text{“reify”}) \qquad \uparrow_\rho : (\mathbb{N} \rightharpoonup \Lambda_\rho) \to [\![\rho]\!] \quad (\text{“reflect”})$$

are defined simultaneously by recursion[4].

$$\begin{aligned} \downarrow_\tau(\mathsf{in}_c(\boldsymbol{b})) &:= c^\infty \downarrow(\boldsymbol{b}), & \uparrow_\tau(r) &:= \mathsf{fam}_\tau(r), \\ \downarrow_\tau(\mathsf{fam}_\tau(r)) &:= r, \\ \downarrow_\tau(\bot) &:= \bot, \\ \downarrow_{\rho\to\sigma}(a)(k) &:= \lambda x_k^\rho \big(\downarrow_\sigma \big(a(\uparrow_\rho(x_k^\infty))\big)(k{+}1)\big), & \uparrow_{\rho\to\sigma}(r)(b) &:= \uparrow_\sigma(r \downarrow_\rho(b)), \\ \downarrow_{\rho\times\sigma}([a,b]) &:= \langle \downarrow_\rho(a), \downarrow_\sigma(b) \rangle, & \uparrow_{\rho\times\sigma}(r) &:= [\uparrow_\rho(r0), \uparrow_\sigma(r1)]. \end{aligned}$$

Note that in $\downarrow_\tau(\mathsf{in}_c(\boldsymbol{b})) := c^\infty \downarrow(\boldsymbol{b})$ we need to refer to $\downarrow$ at higher types.

[2] This is a recursive definition of a family of domains $(\mathcal{I}(\tau))_\tau$ i.e. a least fixed point of certain continuous functions. The theory of (continuous) families of domains and recursive definitions thereof is developed in detail in [3].

[3] From a mathematical point of view it is also possible to take the coalesced sum, but the identification of an undefined object with the total undefined term family is computationaly doubtful.

[4] It is easy to check that the term families stemming from $\downarrow$ are total or the empty term family $\bot$. So if $\bot$ appears in an application, this should always be $\bot$ again.

In the sequel we use (similar to our syntactic convention) the abbreviation ai for $\pi_i(a)$, $i \in \{0,1\}$. We write successive applications of $\uparrow(r)$ to a sequence $\boldsymbol{a} = a_1, \ldots, a_n$ ($a_i \in [\![\rho_i]\!]$ or projection markers 0 or 1) shortly as

$$\uparrow(r)(\boldsymbol{a}) = \uparrow_\rho(r\downarrow(\boldsymbol{a})). \tag{2}$$

In particular, if ρ is a ground type, $\uparrow(r)(\boldsymbol{a}) = \mathsf{fam}_\rho(r\downarrow(\boldsymbol{a}))$ and therefore $\uparrow(r)$ can be understood as a "self-evaluating" interpretation of r.

Without computation rules the definition would be much simpler. It is then possible to define $\mathcal{I}(\tau) := \mathbb{N} \rightharpoonup \Lambda_\tau$, and the functions $\downarrow_\tau$ and $\uparrow_\tau$ would be identities. Then the definition of $\downarrow$ and $\uparrow$ becomes an inductive definition on the types (see [4]).

We will need these functions to define an interpretation of the constants as well as normalization by evaluation itself.

3.4 Predecessor functions

In this section we define for a constructor pattern P^ρ with $\mathsf{FV}(P) = \boldsymbol{x}^{\boldsymbol{\sigma}}$ generalized predecessor functions $\mathsf{gpred}_P : [\![\rho]\!] \to [\![\boldsymbol{\sigma}]\!]$. They are used for the interpretation of the constants in the presence of computation rules.

For analyzing elements of $[\![\tau]\!]$ we define boolean functions $\mathsf{inst?}_P : [\![\rho]\!] \to \mathcal{B}_\perp$ (where $\mathcal{B} = \{\mathtt{tt}, \mathtt{ff}\}$) for every constructor pattern $P^\rho \in \textsc{Constr}_\rho$.

$$\mathsf{inst?}_x(a) := \mathtt{tt}$$

$$\mathsf{inst?}_{cP_1\ldots P_k}(a) := \begin{cases} \bigwedge\!\!\!\bigwedge \mathsf{inst?}_{\boldsymbol{P}}(\boldsymbol{b}) & \text{if } a = \mathsf{in}_c(\boldsymbol{b}) \\ \mathtt{ff} & \text{if } a = \mathsf{in}_d(\boldsymbol{b}) \text{ for some } d \neq c \\ & \text{or } a = \mathsf{fam}_\tau(r) \text{ for some } r \\ \perp & \text{otherwise.} \end{cases}$$

Here $\bigwedge\!\!\!\bigwedge \mathsf{inst?}_{\boldsymbol{P}}(\boldsymbol{b})$ is $\bigwedge_i \mathsf{inst?}_{P_i}(b_i)$ where $\bigwedge$ denotes strict boolean conjunction.

Lemma 14. *Let P, P' be constructor patterns and let $a \in [\![\rho]\!]$. If $\mathsf{inst?}_P(a) = \mathsf{inst?}_{P'}(a) = \mathtt{tt}$, then P and P' are unifiable.*

Proof. Induction on P. □

The generalized predecessor functions

$$\mathsf{gpred}_P : [\![\rho]\!] \to [\![\boldsymbol{\sigma}]\!]$$

are defined inductively for every constructor pattern P^ρ with $\mathsf{FV}(P) = \boldsymbol{x}^{\boldsymbol{\sigma}}$ where the variables are listed from left to right in the order of their occurrences in P.

$$\mathsf{gpred}_x(a) := a,$$

$$\mathsf{gpred}_{c\boldsymbol{P}}(a) := \begin{cases} \mathsf{gpred}_{\boldsymbol{P}}(\boldsymbol{a}) & \text{if } \mathsf{inst?}_{c\boldsymbol{P}}(a) = \mathtt{tt} \text{ and } a = \mathsf{in}_c(\boldsymbol{a}) \\ \perp, \ldots, \perp & \text{otherwise.} \end{cases}$$

Here $\mathsf{gpred}_{\boldsymbol{P}}(\boldsymbol{a})$ denotes the concatenation of the lists $\mathsf{gpred}_{P_i}(b_i)$, and the length of $\perp, \ldots, \perp$ is the number of variables in $c\boldsymbol{P}$.

3.5 Interpretation of the constants

Now we are able to interpret the constants. Notice that $\uparrow$ gives rise to an environment by $x \mapsto \uparrow(x^\infty)$. Let

$$\mathcal{I}(c)(\boldsymbol{a}) := \mathsf{in}_c^{\boldsymbol{\rho}}(\boldsymbol{a}) \quad \text{if } c^{\boldsymbol{\rho}\to\tau} \in \textsc{Constr}.$$

Otherwise $\mathcal{I}(c)$ is defined recursively as follows. If for some computation rule $c\boldsymbol{P} \longmapsto_{\mathsf{comp}} Q$ we have $\bigwedge \mathsf{inst?}_{\boldsymbol{P}}(\boldsymbol{a}) = \mathrm{tt}$, then

$$\mathcal{I}(c)(\boldsymbol{a}) := [\![Q]\!]^{\mathcal{I}}_{[\boldsymbol{x}\mapsto \mathsf{gpred}_{\boldsymbol{P}}(\boldsymbol{a})]}, \quad \text{where } \boldsymbol{x} := \mathsf{FV}(\boldsymbol{P}).$$

If for all computation rules $c\boldsymbol{P} \longmapsto_{\mathsf{comp}} Q$ we have $\bigwedge \mathsf{inst?}_{\boldsymbol{P}}(\boldsymbol{a}) = \mathrm{ff}$, then

$$\mathcal{I}(c)(\boldsymbol{a}) := \begin{cases} [\![N]\!]^{\mathcal{I}}_{[\boldsymbol{x}\mapsto [\![\boldsymbol{L}]\!]^{\mathcal{I}}_{\uparrow}]} & \text{if } \mathsf{sel}_c(\mathsf{ext}(\downarrow(\boldsymbol{a}))) = c\boldsymbol{K} \longmapsto_{\mathsf{rew}} N \\ & \text{and } \mathsf{ext}(\downarrow(\boldsymbol{a})) = \boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}] \\ \uparrow(c^\infty \downarrow(\boldsymbol{a})) & \text{if } \mathsf{sel}_c(\mathsf{ext}(\downarrow(\boldsymbol{a}))) = \mathsf{no\text{-}match}. \end{cases}$$

In all other cases, $\mathcal{I}(c)(\boldsymbol{a}) := \bot$.

Since in the case of computation rules we assumed the left hand sides of these rules to be non-unifiable, lemma 14 guarantees that this is a sound definition.

The usefulness of the computation rules is due to the fact that it is much simpler to compute $\mathsf{gpred}_{\boldsymbol{P}}(\boldsymbol{a})$ than to compute $[\![\boldsymbol{L}]\!]_\uparrow$, where $\mathsf{ext}(\downarrow(\boldsymbol{a})) = \boldsymbol{K}[\boldsymbol{L}]$. This can be seen from the following examples.

Example 15. Let us compare the two possibilities for interpreting constants, either as computation rules or else as rewrite rules. To this end we introduce *iteration types* n, by $0 := \iota$ and $n+1 := n \to n$. Let abst_n be a constructor of type $n \to 0$, and x_n be a variable of iteration type n, e.g. $x_3 \colon ((\iota \to \iota) \to (\iota \to \iota)) \to (\iota \to \iota) \to \iota \to \iota$. We add special constants c_n with the rule

$$c_n(\mathsf{abst}_n x_n) = 0 \tag{3}$$

(0 a constant of type 0). The intention again is to show the difference in efficiency between viewing (3) as computation or as rewrite rule. To see this difference, consider the term $c_n(\mathsf{abst}_n x_n)$ with a variable x_n.

If we view (3) as a rewrite rule, $[\![c_n]\!]$ gets $a := [\![\mathsf{abst}_n x_n]\!]_\uparrow$ as an argument. It computes $\mathsf{ext}(\downarrow(a))$, i.e. the term $\mathsf{abst}_n(\mathsf{nf}(x_n))$. Then it finds $L := \mathsf{nf}(x_n)$, and computes $[\![\mathsf{nf}(x_n)]\!]_\uparrow$. Finally in the environment assigning this value to x_n it computes $[\![0]\!]$.

If however we view (3) as a computation rule, then $[\![c_n]\!]$ again gets $a := [\![\mathsf{abst}_n x_n]\!]_\uparrow$ as an argument. Since $\mathsf{inst?}_{\mathsf{abst}_n x_n}(a) = \mathrm{tt}$, we only need to compute $[\![0]\!]$ in the environment assigning $\mathsf{gpred}_a = \uparrow(x_n^\infty)$ to x_n. This does not involve computation of the long normal form of any variable.

The example shows that normalizing via the rewrite rule may take time exponential in the type level n of x_n, whereas normalizing via the computation rule takes linear time[5]. The reason is simply that the length l_n of the long

[5] We have verified this in the MINLOG system

normal form $\mathsf{nf}(x_n)$ of a variable x_n of iteration type n is $\geq 2^n$. (Proof by induction on n. Step: $l_{n+1} \geq 1 + \sum_{i=0}^{n} l_i \geq 1 + \sum_{i=0}^{n} 2^i = 2^{n+1}$). $\square$

The difference between the two possibilities for interpreting constants can also be seen in the tree example in 2.3(b). In the case of a proper rewrite rule we have

$$\begin{aligned}
\mathcal{I}(\mathcal{O}) &= \mathbb{N} \rightharpoonup \Lambda_{\mathcal{O}}, \\
\mathcal{I}(\textsc{Sup})(a) &= \mathsf{fam}_{\iota}(\textsc{Sup}^{\infty}{\downarrow}(a)) \\
\mathcal{I}(\textsc{Rec})(a, b_0, b_1) &= \begin{cases} b_0 & \text{if } \mathsf{ext}({\downarrow}(a)) = 0 \\ b_1([\![M]\!]_{\uparrow}, g) & \text{if } \mathsf{ext}({\downarrow}(a)) = \textsc{Sup}\, M \text{ and} \\ & g(e) := \mathcal{I}(\textsc{Rec})([\![M]\!]_{\uparrow}(e), b_0, b_1) \\ {\uparrow}(\textsc{Rec}^{\infty}{\downarrow}(a, b_0, b_1)) & \text{if } \mathsf{ext}({\downarrow}(a)) \text{ is defined} \\ & \text{but not } 0 \text{ or } \textsc{Sup}\, N \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

and in case of a computation rule:

$$\begin{aligned}
\mathcal{I}(\mathcal{O}) &= \mathbf{1} + [\mathcal{I}(\iota) \to \mathcal{I}(\mathcal{O})] + (\mathbb{N} \rightharpoonup \Lambda_{\mathcal{O}}), \\
\mathcal{I}(\textsc{Sup})(a) &= in_{\textsc{Sup}}(a) \\
\mathcal{I}(\textsc{Rec})(a, b_0, b_1) &= \begin{cases} b_0 & \text{if } a = in_0(\bot) \\ b_1(f, g) & \text{if } a = in_{\textsc{Sup}}(f) \text{ and} \\ & g(e) := \mathcal{I}(\textsc{Rec})(f(e), b_0, b_1) \\ {\uparrow}(\textsc{Rec}^{\infty}{\downarrow}(a, b_0, b_1)) & \text{if } a \text{ is a term family} \\ \bot & \text{otherwise.} \end{cases}
\end{aligned}$$

Example 16. As mentioned at the end of section 2.4, it makes a difference whether we use the $c\boldsymbol{M}x \longmapsto N$ or the $c\boldsymbol{M} \longmapsto \lambda x N$ version of a rewrite rule. Now we see that in the former case the last argument a forces us to calculate $\mathsf{ext}({\downarrow}(a))$ and then interprete this term again, which may be cumbersome. This can also be verified easily by an example similar to the one above: Let us compare the two rules $c0x_n \longmapsto 0$ and $c0 \longmapsto \lambda x_n 0$. Consider the term $c0x_n$ with a variable x_n of iteration type n. NbE via the first form proceeds as follows. $[\![c]\!]$ gets $a_1 := [\![0]\!]$ and $a_2 := [\![x_n]\!]_{\uparrow} = {\uparrow}(x_n^{\infty})$ as arguments. It computes $\mathsf{ext}({\downarrow}(a_2))$, i.e. the long normal form of x_n. If however we employ NbE via the second form, then $[\![c]\!]$ only gets $a_1 := [\![0]\!]$. It computes $[\![\lambda x 0]\!]$ and applies the result to ${\uparrow}(x_n^{\infty})$; this does not involve computation of any long normal form. $\square$

Lemma 17. *Let P^{ρ} be a constructor pattern with $\mathsf{FV}(P) = \boldsymbol{x}^{\boldsymbol{\sigma}}$, and let $a \in [\![\rho]\!]$. Then for all $\boldsymbol{b} \in [\![\boldsymbol{\sigma}]\!]$*

$$a = [\![P]\!]_{[\boldsymbol{x} \mapsto \boldsymbol{b}]} \quad \textit{iff} \quad \mathsf{inst?}_P(a) = \mathsf{tt} \textit{ and } \mathsf{gpred}_P(a) = \boldsymbol{b}.$$

Proof. Induction on P. If P is a variable, then both sides are equivalent to the statement "$a = b$". Now let $P = c\boldsymbol{P}$. Then $[\![P]\!]_{[\boldsymbol{x} \to \boldsymbol{b}]} = \mathsf{in}_c([\![\boldsymbol{P}]\!]_{[\boldsymbol{x} \to \boldsymbol{b}]})$. Assume

$a = [\![P]\!]_{[\boldsymbol{x} \mapsto \boldsymbol{b}]}$. Then $a = \mathsf{in}_c(\boldsymbol{a})$ where $\boldsymbol{a} = [\![\boldsymbol{P}]\!]_{[\boldsymbol{x} \to \boldsymbol{b}]}$. Hence $\bigwedge \mathsf{inst?}_{\boldsymbol{P}}(\boldsymbol{a}) = \mathsf{tt}$ and $\mathsf{gpred}_{\boldsymbol{P}}(\boldsymbol{a}) = \boldsymbol{b}$, by IH. Therefore $\mathsf{inst?}_{c\boldsymbol{P}}(a) = \mathsf{tt}$ and $\mathsf{gpred}_{c\boldsymbol{P}}(a) = \mathsf{gpred}_{\boldsymbol{P}}(\boldsymbol{a}) = \boldsymbol{b}$. For the converse assume $\mathsf{inst?}_{c\boldsymbol{P}}(a) = \mathsf{tt}$ and $\mathsf{gpred}_{c\boldsymbol{P}}(a) = \boldsymbol{b}$. Then $a = \mathsf{in}_c(\boldsymbol{a})$, $\bigwedge \mathsf{inst?}_{\boldsymbol{P}}(\boldsymbol{a}) = \mathsf{tt}$ and moreover $\mathsf{gpred}_{\boldsymbol{P}}(\boldsymbol{a}) = \mathsf{gpred}_{c\boldsymbol{P}}(a) = \boldsymbol{b}$. Hence $[\![\boldsymbol{P}]\!]_{[\boldsymbol{x} \to \boldsymbol{b}]} = \boldsymbol{a}$, by IH. Hence $a = \mathsf{in}_c(\boldsymbol{a}) = [\![P]\!]_{[\boldsymbol{x} \to \boldsymbol{b}]}$. □

Lemma 18. *For every rule $c\boldsymbol{P} \longmapsto_{\mathsf{comp}} Q$ and every environment ξ*

$$[\![c\boldsymbol{P}]\!]_\xi = [\![Q]\!]_\xi.$$

Proof. Let $\mathsf{FV}(\boldsymbol{P}) = \boldsymbol{x}$. By the previous lemma, $\bigwedge \mathsf{inst?}_{\boldsymbol{P}}([\![\boldsymbol{P}]\!]_\xi) = \mathsf{tt}$ and $\mathsf{gpred}_{\boldsymbol{P}}([\![\boldsymbol{P}]\!]_\xi) = \xi(\boldsymbol{x})$. Hence, since c is not a constructor, $[\![c\boldsymbol{P}]\!]_\xi = \mathcal{I}(c)([\![\boldsymbol{P}]\!]_\xi) = [\![Q]\!]_\xi$. □

3.6 Correctness of normalization by evaluation

We say that normalization by evaluation is correct if $M \longrightarrow Q$ implies that $\downarrow([\![M]\!]_\uparrow)(k) =_\alpha Q$ for $k > \mathsf{FV}(M)$. If we have preservation of values (i.e. that conversion does not change the meaning (i.e. semantics) of a term), then this is easy to prove, even in the stronger from $\downarrow([\![M]\!]_\uparrow) = Q^\infty$ (by induction on terms M in long normal form). But in general, preservation of values is not guaranteed.

Example 19. In the presence of higher-type variables it is possible that rewrite rules do not preserve the value. Let c be a constant of type $(\tau \to \tau) \to \tau \to \tau$ with the single rule $cx \longmapsto x$, considered as a proper rewrite rule. Choose $\mathsf{id}' \in [\![\tau \to \tau]\!]^{\mathcal{I}} = [\mathcal{I}(\tau) \to \mathcal{I}(\tau)]$ with $\mathsf{id}' \neq \mathsf{id}$ but $\mathsf{id}'(\mathsf{fam}_\tau(x^\infty)) = \mathsf{fam}_\tau(x^\infty)$ for every variable $x \in \Lambda_\tau$. Now preservation of value would require $[\![cx]\!]_\xi = [\![x]\!]_\xi$ for every environment ξ, so in particular for $\xi(x) = \mathsf{id}'$. But because of the reference to $\downarrow_{\tau\to\tau}$ in the definition of $\mathcal{I}(c)$ we have $\mathcal{I}(c)(\mathsf{id}') = \mathsf{id}$. To see this, observe that $\downarrow_{\tau\to\tau}(\mathsf{id}')(k) = \lambda x_k.(\mathsf{id}'(\uparrow(\mathsf{fam}_\tau(x_k^\infty)))(k+1)) = \lambda x_k x_k$, hence $\mathsf{ext}(\downarrow(\mathsf{id}')) = \lambda x_1 x_1$ and therefore $\mathcal{I}(c)(\mathsf{id}') = [\![\lambda x_1 x_1]\!]^{\mathcal{I}} = \mathsf{id}$. □

Example 20. In the non-confluent rewrite system considered in example 7 we had $d2 \longrightarrow \lambda x2$. However, for the interpretation of $d2$ and $\lambda x2$ we get different values: $[\![d2]\!](3^\infty) = \mathcal{I}(d)(2^\infty)(3^\infty) = 3^\infty$ since $\mathsf{sel}_d(2,3) = dx3 \longmapsto 3$, but $[\![\lambda x2]\!](3^\infty) = 2^\infty$. □

However, we still have correctness of normalization by evaluation.

Theorem 21. *a. If $M \longrightarrow Q$, then $\downarrow([\![M]\!]_\uparrow)(k) = Q^\infty(k)$ for $k > \mathsf{FV}(M)$.*

b. Assume $M \longrightarrow_s Q$. Then $\downarrow([\![M]\!]_\uparrow)(k) = Q^\infty(k)$ for $k > \mathsf{FV}(M)$, and if $\mathsf{inst?}_P([\![M]\!]_\uparrow) = \mathsf{tt}$ for some constructor pattern P, then there are terms $\boldsymbol{L}$ such that $M = P_{\boldsymbol{x}}[\boldsymbol{L}]$ and $\mathsf{gpred}_P([\![M]\!]_\uparrow) = [\![\boldsymbol{L}]\!]_\uparrow$[6].

c. If $M \longrightarrow_w Q$, then $[\![M]\!]_\uparrow = [\![Q]\!]_\uparrow$.

[6] Moreover, if we assume strong uniformity of all sel_c-functions (as we implicitely did in [4]), then $\downarrow([\![M]\!]_\uparrow) = Q^\infty$ holds in parts a and b.

Example 22. Without strong uniformity of the select functions the stronger claim $\downarrow([\![M]\!]_\uparrow) = Q^\infty$ for $M \longrightarrow Q$ is not true. For a counterexample consider the single rewrite rule $cxx \longmapsto_{\mathsf{rew}} d$ with x, d of ground type τ. Then $cx_0x_1 \longrightarrow cx_0x_1$ by PassApp, hence $cx_0 \longrightarrow \lambda x_1.cx_0x_1$ by Eta. We obtain

$$\begin{aligned}\downarrow([\![cx_0]\!]_\uparrow)(0) &= \lambda x_0\big(\downarrow_\tau([\![cx_0]\!]_\uparrow(\uparrow_\tau(x_0^\infty)))(1)\big)\\ &= \lambda x_0\big(\downarrow_\tau(\mathcal{I}(c)(\mathsf{fam}_\tau(x_0^\infty), \mathsf{fam}_\tau(x_0^\infty)))(1)\big)\end{aligned}$$

Now $\mathsf{ext}(\downarrow_\tau(\mathsf{fam}_\tau(x_0^\infty))) = \mathsf{ext}(x_0^\infty) = x_0$ and $\mathsf{sel}_c(x_0, x_0) = cxx \longmapsto_{\mathsf{rew}} d$, hence

$$\begin{aligned}&= \lambda x_0\big(\downarrow_\tau(\mathcal{I}(d)))(1)\big)\\ &= \lambda x_0\big(\downarrow_\tau(\mathsf{fam}_\tau(d^\infty))(1)\big)\\ &= \lambda x_0 d.\end{aligned}$$

However, $(\lambda x_1.cx_0x_1)^\infty(0) = \lambda x_0.cx_0x_0$. □

Proof of the theorem. By simultaneous induction on the height of the derivation of $M \longrightarrow Q$ resp. $M \longrightarrow_w Q$ and $M \longrightarrow_s Q$. For brevity we leave out the rules concerning product types, since their treatment does not bring up any new issues. Note that the second statement about $\longrightarrow_s$ holds if P is a variable, therefore we assume in the sequel that P is not a variable and thus of ground type.

Case Split.
$$\frac{M \longrightarrow_w^* N \quad N \longrightarrow_s Q}{M \longrightarrow Q}$$

By IH for the left premise $[\![M]\!]_\uparrow = [\![N]\!]_\uparrow$, thus we can apply the IH to the right premise to infer $\downarrow([\![M]\!]_\uparrow)(k) = \downarrow([\![N]\!]_\uparrow)(k) = Q^\infty(k)$ for $k > \mathsf{FV}(M)$.

Case Eta.
$$\frac{My \longrightarrow Q}{M \longrightarrow_s \lambda y Q} \quad \text{for } y \notin \mathsf{FV}(M).$$

Let $k > \mathsf{FV}(M)$. Then by lemma 6 $Mx_k \longrightarrow Q_y[x_k]$ with a derivation of the same height. Hence

$$\begin{aligned}\downarrow([\![M]\!]_\uparrow)(k) &= \lambda x_k\big(\downarrow([\![M]\!]_\uparrow(\uparrow(x_k^\infty)))(k+1)\big)\\ &= \lambda x_k\big(\downarrow([\![Mx_k]\!]_\uparrow)(k+1)\big)\\ &= \lambda x_k\big(Q_y[x_k]^\infty(k+1)\big) && \text{by IH, since } k+1 > \mathsf{FV}(Mx_k)\\ &= (\lambda y Q)^\infty(k).\end{aligned}$$

The additional claim holds since M is not of ground type.

Case VarApp.
$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M}'}{x\boldsymbol{M} \longrightarrow_s x\boldsymbol{M}'}$$

with $x\boldsymbol{M}$ of ground type. We have

$$[\![x\boldsymbol{M}]\!]_\uparrow = \uparrow(x^\infty)([\![\boldsymbol{M}]\!]_\uparrow) = \mathsf{fam}_\tau(x^\infty \downarrow([\![\boldsymbol{M}]\!]_\uparrow))$$

by (2), hence

$$\downarrow([\![x\boldsymbol{M}]\!]_\uparrow)(k) = \big(x^\infty\downarrow([\![\boldsymbol{M}]\!]_\uparrow)\big)(k) = x(\boldsymbol{M}')^\infty(k) = (x\boldsymbol{M}')^\infty(k)$$

by IH. The second claim holds since

$$\mathsf{inst?}_P([\![x\boldsymbol{M}]\!]_\uparrow) = \mathsf{inst?}_P(\mathsf{fam}_\tau\big(x^\infty\downarrow([\![\boldsymbol{M}]\!]_\uparrow)\big)) = \mathrm{ff}.$$

Case BETA.

$$(\lambda x M)N\boldsymbol{P} \longrightarrow_w M_x[N]\boldsymbol{P}$$

Use $[\![(\lambda x M)N\boldsymbol{P}]\!]_\uparrow = [\![M_x[N]\boldsymbol{P}]\!]_\uparrow$, which holds in every model of the λ-calculus.

Case COMP.

$$c\boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N} \longrightarrow_w Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N} \quad \text{if } c\boldsymbol{P} \longmapsto_{\mathsf{comp}} Q.$$

Then

$$\begin{aligned}
[\![c\boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}]\!]_\uparrow &= [\![c\boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]]\!]_\uparrow[\![\boldsymbol{N}]\!]_\uparrow \\
&= [\![c\boldsymbol{P}]\!]_{[\boldsymbol{x}\mapsto[\![\boldsymbol{L}]\!]_\uparrow]}[\![\boldsymbol{N}]\!]_\uparrow && \text{by the substitution lemma} \\
&= [\![Q]\!]_{[\boldsymbol{x}\mapsto[\![\boldsymbol{L}]\!]_\uparrow]}[\![\boldsymbol{N}]\!]_\uparrow && \text{by lemma 18} \\
&= [\![Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}]\!]_\uparrow.
\end{aligned}$$

Case ARG.

$$\frac{\boldsymbol{M} \longrightarrow_w^* \boldsymbol{Q}}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_w c\boldsymbol{Q}\boldsymbol{N}}$$

Then $[\![c\boldsymbol{M}\boldsymbol{N}]\!]_\uparrow = \mathcal{I}(c)[\![\boldsymbol{M}]\!]_\uparrow[\![\boldsymbol{N}]\!]_\uparrow = \mathcal{I}(c)[\![\boldsymbol{Q}]\!]_\uparrow[\![\boldsymbol{N}]\!]_\uparrow = [\![c\boldsymbol{Q}\boldsymbol{N}]\!]_\uparrow$ by IH.

Case REW.

$$\frac{\boldsymbol{M} \longrightarrow_s \boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}]}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_w Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}} \quad \text{if } c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q$$

where $\mathsf{sel}_c(\boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}]) = c\boldsymbol{K} \longmapsto_{\mathsf{rew}} Q$ and $c\boldsymbol{M}$ is not the instance of a computation rule.

We show that $\bigwedge \mathsf{inst?}_{\boldsymbol{P}}([\![\boldsymbol{M}]\!]_\uparrow) = \mathrm{ff}$ for all computation rules $c\boldsymbol{P} \longmapsto_{\mathsf{comp}} Q$. So assume first that $\bigwedge \mathsf{inst?}_{\boldsymbol{P}}([\![\boldsymbol{M}]\!]_\uparrow) = \mathrm{tt}$. Then by IH there are $\boldsymbol{L}$ such that $\boldsymbol{M} = \boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]$ (note that $\boldsymbol{P}$ is linear), so $c\boldsymbol{M}$ would be an instance of a computation rule, which contradicts the assumption. It remains to show that $\mathsf{inst?}_{P_i}([\![M_i]\!]_\uparrow) \neq \bot$ for all i. But if we had $= \bot$ for some i, then $[\![M_i]\!]_\uparrow = \bot$ in contrast to $\downarrow([\![\boldsymbol{M}]\!]_\uparrow)(k) = \boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}]^\infty(k)$, which holds by IH. Moreover, $\mathsf{sel}_c(\mathsf{ext}(\downarrow([\![\boldsymbol{M}]\!]_\uparrow))) = \mathsf{sel}_c(\boldsymbol{K}_{\boldsymbol{x}}[\boldsymbol{L}]) = c\boldsymbol{K} \longmapsto Q$. This gives us the necessary information about $\mathcal{I}(c)$.

$$\begin{aligned}
[\![c\boldsymbol{M}\boldsymbol{N}]\!]_\uparrow &= \mathcal{I}(c)([\![\boldsymbol{M}]\!]_\uparrow)([\![\boldsymbol{N}]\!]_\uparrow) \\
&= [\![Q]\!]_{[\boldsymbol{x}\mapsto[\![\boldsymbol{L}]\!]_\uparrow]}([\![\boldsymbol{N}]\!]_\uparrow) && \text{by definition of } \mathcal{I}(c) \\
&= [\![Q_{\boldsymbol{x}}[\boldsymbol{L}]]\!]_\uparrow([\![\boldsymbol{N}]\!]_\uparrow) && \text{by the substitution lemma} \\
&= [\![Q_{\boldsymbol{x}}[\boldsymbol{L}]\boldsymbol{N}]\!]_\uparrow.
\end{aligned}$$

Case PassApp.
$$\frac{\boldsymbol{M} \longrightarrow_s \boldsymbol{M}' \quad \boldsymbol{N} \longrightarrow \boldsymbol{N}'}{c\boldsymbol{M}\boldsymbol{N} \longrightarrow_s c\boldsymbol{M}'\boldsymbol{N}'}$$
where $c\boldsymbol{M}$ is not an instance of a computation rule, $\mathsf{sel}_c(\boldsymbol{M'}) = \mathsf{no\text{-}match}$ and $c\boldsymbol{M}\boldsymbol{N}$ of ground type.

Let us first consider the case where c is a constructor (then $\boldsymbol{N}$ is empty). We obtain
$$[\![c\boldsymbol{M}]\!]_\uparrow = \mathcal{I}(c)([\![\boldsymbol{M}]\!]_\uparrow) = \mathsf{in}_c([\![\boldsymbol{M}]\!]_\uparrow),$$
hence by definition of $\downarrow$ and the IH
$$\downarrow([\![c\boldsymbol{M}]\!]_\uparrow)(k) = \bigl(c^\infty{\downarrow}([\![\boldsymbol{M}]\!]_\uparrow)\bigr)(k) = \bigl(c\boldsymbol{M}'\bigr)^\infty(k).$$
This is the first claim. Now assume $P = c\boldsymbol{P}$ and $\mathsf{inst?}_P([\![c\boldsymbol{M}]\!]_\uparrow) = \mathsf{tt}$. Then $\bigwedge\mathsf{inst?}_{\boldsymbol{P}}([\![\boldsymbol{M}]\!]_\uparrow) = \mathsf{tt}$ and by IH there are $\boldsymbol{L}$ such that $\boldsymbol{M} = \boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]$ (note that $c\boldsymbol{P}$ is linear), hence $c\boldsymbol{M} = c\boldsymbol{P}_{\boldsymbol{x}}[\boldsymbol{L}]$.

If c is not a constructor, we have $\bigwedge\mathsf{inst?}_{\boldsymbol{P}}([\![\boldsymbol{M}]\!]_\uparrow) = \mathsf{ff}$ (with the same argument as in Rew) and
$$\mathsf{sel}_c(\mathsf{ext}({\downarrow}([\![\boldsymbol{M}]\!]_\uparrow))) = \mathsf{sel}_c(\mathsf{ext}\bigl((\boldsymbol{M}')^\infty\bigr) = \mathsf{sel}_c(\boldsymbol{M}') = \mathsf{no\text{-}match},$$
since by IH ${\downarrow}([\![\boldsymbol{M}]\!]_\uparrow)(k) = (\boldsymbol{M}')^\infty(k)$ for $k > \mathsf{FV}(\boldsymbol{M})$. Now we can compute $[\![c\boldsymbol{M}\boldsymbol{N}]\!]_\uparrow$.
$$\begin{aligned}[\![c\boldsymbol{M}\boldsymbol{N}]\!]_\uparrow &= \mathcal{I}(c)([\![\boldsymbol{M}]\!]_\uparrow)([\![\boldsymbol{N}]\!]_\uparrow)\\ &= {\uparrow}\bigl(c^\infty{\downarrow}([\![\boldsymbol{M}]\!]_\uparrow)\bigr)([\![\boldsymbol{N}]\!]_\uparrow)\\ &= \mathsf{fam}_\tau\bigl(c^\infty{\downarrow}([\![\boldsymbol{M},\boldsymbol{N}]\!]_\uparrow)\bigr) \quad \text{by (2)}\end{aligned}$$
hence
$$\begin{aligned}{\downarrow}([\![c\boldsymbol{M}\boldsymbol{N}]\!]_\uparrow)(k) &= \bigl(c^\infty{\downarrow}([\![\boldsymbol{M},\boldsymbol{N}]\!]_\uparrow)\bigr)(k)\\ &= \bigl(c\boldsymbol{M}'\boldsymbol{N}'\bigr)^\infty(k) \qquad \text{by IH.}\end{aligned}$$
The second claim holds trivially again. □

Corollary 23. *If M is strongly normalizable, then $M \longrightarrow \mathsf{nf}(M)$, and therefore the (long) normal form $\mathsf{nf}(M)$ of M can be obtained as ${\downarrow}\bigl([\![M]\!]_\uparrow\bigr)(k) =_\alpha \mathsf{nf}(M)$, for any $k > \mathsf{FV}(M)$.*

Proof. Let M be strongly normalizable. Then $M \longrightarrow Q$ for some Q by lemma 5, and Q is the long normal form $\mathsf{nf}(M)$ of M by lemma 4. By the theorem ${\downarrow}\bigl([\![M]\!]_\uparrow\bigr)(k) = \mathsf{nf}(M)^\infty(k)$ for any $k > \mathsf{FV}(M)$. Now use lemma 9b. □

Another immediate corollary is that the $\longrightarrow$-relation is indeed a (partial) function: If $M \longrightarrow Q$ and $M \longrightarrow P$, then $P^\infty(k) = {\downarrow}\bigl([\![M]\!]_\uparrow\bigr)(k) = Q^\infty(k)$ for $k > \mathsf{FV}(M)$, hence $P =_\alpha Q$ by lemma 10.

Acknowledgements. The present work has benefitted considerably from ideas of Felix Joachimski and Ralph Matthes concerning strategies for normalization proofs, including η-expansion and primitive recursion. In particular, the idea to employ the inductive definition of the relation $M \longrightarrow Q$ is essentially due to them. We also want to thank Holger Benl for illuminating discussions, and an anonymous referee for his helpful comments.

References

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *CTCS'95, Cambridge*, volume 953 of *LNCS*, pages 182–199. Springer Verlag, Berlin, Heidelberg, New York, 1995.

[2] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolfgang Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*, pages 41–71. Kluwer Academic Publishers, Dordrecht, 1998.

[3] Ulrich Berger. Continuous functionals of dependent and transfinite types. Habilitationsschrift, Mathematisches Institut der Universität München, 1997.

[4] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, volume 1546 of *LNCS*, pages 117–137. Springer Verlag, Berlin, Heidelberg, New York, 1998.

[5] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In R. Vemuri, editor, *Proceedings 6'th Symposium on Logic in Computer Science (LICS'91)*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[6] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:73–94, 1997.

[7] Roy L. Crole. *Categories for Types.* Cambridge University Press, 1993.

[8] Olivier Danvy. Pragmatics of type-directed partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *LNCS*, pages 73–94. Springer Verlag, Berlin, Heidelberg, New York, 1996.

[9] Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Math.*, 34:381–392, 1972.

[10] Andrzej Filinski. A semantic account of type-directed partial evaluation. In *Principles and Practice of Declarative Programming 1999*, volume 1702 of *LNCS*, pages 378–395. Springer Verlag, Berlin, Heidelberg, New York, 1999. http://www.brics.dk/ andrzej/papers/.

[11] Jim Lambek and Phil Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[12] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, 1960.

[13] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.