Big-Step Normalisation

THORSTEN ALTENKIRCH and JAMES CHAPMAN
School of Computer Science,
University of Nottingham, UK

Abstract

Traditionally, decidability of conversion for typed λ-calculi is established by showing that small-step reduction is confluent and strongly normalising. Here we investigate an alternative approach employing a recursively defined normalisation function which we show to be terminating and which reflects and preserves conversion. We apply our approach to the simply-typed λ-calculus with explicit substitutions and $\beta\eta$-equality, a system which is not strongly normalising. We also show how the construction can be extended to System T with the usual β-rules for the recursion combinator. Our approach is practical, since it does verify an actual implementation of normalisation which, unlike normalisation by evaluation, is first order. An important feature of our approach is that we are using logical relations to establish equational soundness (identity of normal forms reflects the equational theory), instead of the usual syntactic reasoning using the Church-Rosser property of a term rewriting system.

1 Introduction

Traditionally, decidability of conversion for typed λ-calculi is established by showing that small-step reduction is confluent and strongly normalising, e.g. see (Girard *et al.*, 1989) where this approach is applied to the simply-typed λ-calculus, System F and System T. In fact, decidability is not the only corollary of strong normalisation, we can reason using the structure of normal forms and show for example that certain types are not inhabited.

The small-step approach does not extend easily to stronger conversion relations, e.g. η-conversion. η-reduction preserves strong normalisation, but η-expansion obviously doesn't. On the other hand η-expansion is preferable because normal terms are in constructor form (i.e. λ-abstractions). This issue can by addressed by careful modification of the reduction relation (Jay & Ghani, 1995). A more serious issue arises when introducing substitution as an explicit operation (Abadi *et al.*, 1990) — this is better, because it treats substitution in the same way as other operators such as application. It was hoped that the small-step semantics for substitution would mix well with β-reduction — this hope was dashed by Melliès's observation that $\sigma\beta$-reduction is not strongly normalising (Melliès, 1995).

All these issues can be addressed by ingenious modifications of the small-step semantics. However, it is doubtful that anybody would actually want to implement a normalisation function by laboriously applying one-step reductions to a term. [1]

We observe that normalisation can be expressed by the following specification: we introduce a notion of normal forms indexed over context Γ and type σ: $\mathsf{Nf}\,\Gamma\,\sigma$ which can be embedded back into terms $\mathsf{Tm}\,\Gamma\,\sigma$. We assume that we can realize a function **nf** which for any $t : \mathsf{Tm}\,\Gamma\,\sigma$ calculates a normal form $\mathbf{nf}\,t : \mathsf{Nf}\,\Gamma\,\sigma$, such that the following properties[2] hold:

soundness Normalisation takes convertible terms to identical normal forms

$$\frac{t \simeq t'}{\mathbf{nf}\,t = \mathbf{nf}\,t'}$$

completeness Terms are convertible to their normal forms

$$t \simeq \mathbf{nf}\,t$$

As a consequence we obtain that convertibility corresponds to having the same normal form:

$$t \simeq u \iff \mathbf{nf}\,t = \mathbf{nf}\,u$$

Since the equality of normal forms is obviously decidable, we have that conversion is decidable. Additionally, we would like that the notion of normal form contains no redundant elements — and hence we can establish additional properties by induction over the structure of normal forms. We can capture this property by additionally demanding:

stability Normalisation is stable on normal forms.

$$\frac{n : \mathsf{Nf}\,\Gamma\,\sigma}{\mathbf{nf}\,n = n}$$

Strong normalisation gives rise to one way to implement this specification. An alternative is *normalisation by evaluation*, a technique pioneered in (Berger & Schwichtenberg, 1991). Normalisation by evaluation exploits a complete model construction where the evaluation function can be inverted. The composition of the evaluation function and its inverse gives rise to a normalisation function. This function can be executed since all the steps take place in a constructive metatheory. Normalisation by evaluation overcomes many of the shortcomings of the small-step approach. Indeed, decidability for strong equality of λ-calculus with coproducts was first shown using normalisation by evaluation (Altenkirch *et al.*, 2001; Balat, 2002). More recently a normalising small-step semantics was introduced by (Lindley, 2007), but strong normalisation is still open. Moreover, normalisation by evaluation is practical, it has been used and has been used in the actual implementation of Schwichtenberg's Minlog system.

Here we investigate yet another alternative: big-step normalisation. This is in

[1] This criticism applies only to small-step term rewriting, clearly it is computationally sensible to model the computation of a normal form by performing small steps of an abstract machine.

[2] Our terminology here is motivated by the view that the normal forms form a syntactic model of the calculus.

some way the most naive approach to normalisation: we use an environment machine, implemented as a functional program, to reduce programs to values and apply this method recursively to quote values as normal forms. We apply this approach here to $\lambda^{\beta\eta\sigma}$, simply-typed λ-calculus with explicit substitutions, a calculus which is difficult to capture using small-step reduction. Unlike normalisation by evaluation our approach is first order, we do not need higher order functions in any essential[3] way to implement normalisation by evaluation assumes that we already have a means to evaluate higher order programs, i.e. λ-terms.

Big-step normalisation shares the logical structure of small-step normalisation. The normalisation function is specified as an inductive relation using only first order means. This relation is executable, indeed it is derived from a recursive functional program, and then shown to be terminating. Unlike Normalisation by evaluation there is a strict separation between the first order structure of the program and the higher order reasoning needed to establish termination. [4]

Related work

(Levy, 2001) uses Tait's method to show normalisation for the big-step semantics of a simple programming language. In our conference paper we have developed big-step normalisation for a combinatory version of System T (Altenkirch & Chapman, 2006). T. Coquand uses a variant of big-step reduction to normalise terms in Type Theory (Coquand, 1991), however, he exploits a model of the untyped λ-calculus to implement normalisation, which is not necessary in our approach. Our work is closely related to C. Coquand's formalisation of another variant of simply-typed λ-calculus with substitutions (Coquand, 2002) — the main difference to the present work is that she uses normalisation by evaluation.

Type Theory as a metalanguage

We use Type Theory as a metalanguage, hence when we define a function we can also run it as a functional program. However, since we do not exploit propositions as types in any essential way, our development can be understood as taking place in naive set theory.

Our notation is very much inspired by the Epigram system (McBride, 2005a), which we have used together with Agda (Norell, 2007a) for a formalisation of the material presented here (Chapman, 2007).

We use $\star$ for the type of small types (or sets), and Prop as the type of propositions. We will not use proofs of propositions to make choices, i.e. we assume a proof-irrelevant universe of propositions. We present inductively defined families, types

[3] We may still use higher order functions like **map** to reuse code, but this use of higher order functions can be easily eliminated by expanding the definitions.

[4] Note that we do not attempt to give a normalisation argument which can be formalized in first-order arithmetic, such as the one given in (David, 2001). Indeed, we show in section 7 that our construction easily extends to System T, whose normalisation proof certainly cannot be formalized in first-order arithmetic.

and predicates by giving the constructors in a natural deduction style, inspired by the syntax of the Epigram system[5] . We construct functions and proofs by structural recursion over inductive definitions which, using the tactics implemented in Epigram, are reducible to basic Type Theory using only standard combinators.

As in Epigram and other implementations of Type Theory we hide arguments and types which can be inferred from the context to make the code more readable. If we want to make implicit arguments explicit we put them in subscript position. As an example consider our presentation of an inductive definitions of the set of natural numbers:

$$\frac{}{\mathsf{Nat} \ : \ \star} \quad \frac{\text{where}}{} \quad \frac{}{\mathsf{zero} \ : \ \mathsf{Nat}} \quad \frac{n \ : \ \mathsf{Nat}}{\mathsf{suc}\, n \ : \ \mathsf{Nat}}$$

and the family of finite types:

$$\frac{n \ : \ \mathsf{Nat}}{\mathsf{Fin}\, n \ : \ \star} \quad \frac{\text{where}}{} \quad \frac{}{\mathsf{fzero} \ : \ \mathsf{Fin}\,(\mathsf{suc}\, n)} \quad \frac{i \ : \ \mathsf{Fin}\, n}{\mathsf{fsuc}\, i \ : \ \mathsf{Fin}\,(\mathsf{suc}\, n)}$$

Note that we omit the declaration of $n \ : \ \mathsf{Nat}$ as an implicit argument to the constructors fzero and fsuc, because it can be automatically inferred by the system. More details and examples can be found in the Epigram tutorial (McBride, 2005b).

Overview of the paper

We introduce a simply-typed λ-calculus with explicit substitution and $\beta\eta$-equality in section 2. We then implement a recursive normalisation function in partial Type Theory in section 3. Using a technique introduced in (Bove & Capretta, 2001) we use a relational presentation, i.e., a *big-step reduction* relation of the partial functions in total Type Theory to be able to characterize the graph of our normalisation function in section 4. Using a variant of strong computability[6] (Tait, 1967), incorporating Kripke logical predicates, we then show that our partial normalisation function terminates and returns a result convertible to the input in section 5. It remains to show soundness, we do this using Kripke logical relations in section 6. We show that this approach is easily extensible to System T with β-rules for the recursion combinator in section 7. We finish with general observations about our approach and sketch future work (section 8).

2 Simply-typed λ-calculus with explicit substitutions

We present here the simply-typed λ-calculus with explicit substitutions, much in the spirit of the λ^{σ}-calculus (Abadi *et al.*, 1990). This approach avoids the special status of substitution which traditionally, unlike other term formers, is defined by recursion over the syntax. In our presentation, substitution is a term former like

[5] If you are looking at a polychrome version of this paper, you will notice that we also follow Epigram's colour conventions for types, constructors, **functions** and *variables*.

[6] Also called strong reducibility. The strength here refers to the necessary strengthening of the induction hypothesis, and to strong normalisation.

any other, with a set of equationally specified properties. We also diverge from the conventional strategy of defining pre-terms first and then to introduce a typing judgement. Instead we directly present the family of well-typed terms as in inductively defined family. We are, after all, only interested in the well-typed terms.

Syntax

The inductive definition of the set of types $\mathsf{Ty} : \star$ with one base type and contexts $\mathsf{Con} : \star$ as backwards written lists of types are straightforward:

$$\frac{}{\bullet : \mathsf{Ty}} \quad \frac{\sigma : \mathsf{Ty} \quad \tau : \mathsf{Ty}}{(\sigma \to \tau) : \mathsf{Ty}} \qquad \frac{}{\varepsilon : \mathsf{Con}} \quad \frac{\Gamma : \mathsf{Con} \quad \sigma : \mathsf{Ty}}{(\Gamma ; \sigma) : \mathsf{Con}}$$

We define inductive families of well-typed terms and substitutions mutually.

$$\frac{\Gamma : \mathsf{Con} \quad \sigma : \mathsf{Ty}}{\mathsf{Tm}\, \Gamma\, \sigma : \star} \quad \frac{\Gamma, \Delta : \mathsf{Con}}{\mathsf{Subst}\, \Gamma\, \Delta : \star}$$

The syntax of terms uses categorical combinators which subsumes variables. There is a term $\varnothing$ which refers to the last variable in the context and $t[\vec{t}]$ is the application of an explicit substitution to a term. Variables other than the last can be constructed by combining $\varnothing$ with weakening substitutions $\uparrow_\sigma$, which corresponds to $+1$ in a de Bruijn representation.

$$\frac{}{\varnothing : \mathsf{Tm}\, (\Gamma ; \sigma)\, \sigma} \quad \frac{t : \mathsf{Tm}\, \Delta\, \sigma \quad \vec{t} : \mathsf{Subst}\, \Gamma\, \Delta}{t[\vec{t}] : \mathsf{Tm}\, \Gamma\, \sigma}$$

$$\frac{t : \mathsf{Tm}\, (\Gamma ; \sigma)\, \tau}{\lambda_\sigma t : \mathsf{Tm}\, \Gamma\, (\sigma \to \tau)} \quad \frac{t : \mathsf{Tm}\, \Gamma\, (\sigma \to \tau) \quad u : \mathsf{Tm}\, \Gamma\, \sigma}{t\, u : \mathsf{Tm}\, \Gamma\, \tau}$$

As an example we represent the λ-term implementing the S combinator (given $\sigma, \tau, \rho : \mathsf{Ty}$) :

$$\vdash \lambda f. \lambda g. \lambda x. f\, x\, (g\, x) : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \tau$$

as

$$\lambda(\lambda(\lambda((\varnothing[\uparrow_{\sigma \to \tau}][\uparrow_\sigma])\, \varnothing\, ((\varnothing[\uparrow_\sigma])\, \varnothing)))) : \mathsf{Tm}\, \varepsilon\, ((\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \tau)$$

Our syntax for substitutions uses the standard categorical combinators: id_Γ the identity substitution, $\vec{t} \circ \vec{u}$ composition of substitutions, $\vec{t}; t$ extension of a substitution and $\uparrow_\sigma$ weakening or projection.

$$\frac{}{\mathsf{id}_\Gamma : \mathsf{Subst}\, \Gamma\, \Gamma} \quad \frac{\vec{t} : \mathsf{Subst}\, \Gamma\, \Delta \quad \vec{u} : \mathsf{Subst}\, \Sigma\, \Gamma}{\vec{t} \circ \vec{u} : \mathsf{Subst}\, \Sigma\, \Delta}$$

$$\frac{\vec{t} : \mathsf{Subst}\, \Gamma\, \Delta \quad t : \mathsf{Tm}\, \Gamma\, \sigma}{(\vec{t}; t) : \mathsf{Subst}\, \Gamma\, (\Delta ; \sigma)} \quad \frac{}{\uparrow_\sigma : \mathsf{Subst}\, (\Gamma ; \sigma)\, \Gamma}$$

As a special case we can derive substitution of the last variable by a term: given $t : \mathsf{Tm}\, (\Gamma ; \sigma)\, \tau$ and $u : \mathsf{Tm}\, \Gamma\, \tau$, we represent t with $\varnothing$ substituted by u as $t[u] = t[\mathsf{id}_\Gamma ; u] : \mathsf{Tm}\, \Gamma\, \sigma$.

Equational Theory

We define weak conversion $\simeq_{\mathsf{w}\sigma}$ and strong (or $\beta\eta$) conversion for terms and substitutions $\simeq_{\beta\eta\sigma}$. Each of them is defined mutually for terms and substitutions:

$$\frac{t, u \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\begin{array}{c} t \;\simeq_{\beta\eta\sigma}\; u \;:\; \mathsf{Prop} \\ t \;\simeq_{\mathsf{w}\sigma}\; u \;:\; \mathsf{Prop}\end{array}} \qquad \frac{\vec{t}, \vec{t'} \;:\; \mathsf{Subst}\,\Gamma\,\Delta}{\begin{array}{c} \vec{t} \;\simeq_{\beta\eta\sigma}\; \vec{t'} \;:\; \mathsf{Prop} \\ \vec{t} \;\simeq_{\mathsf{w}\sigma}\; \vec{t'} \;:\; \mathsf{Prop}\end{array}}$$

Weak conversion corresponds to combinatorial equality and excludes the η-rule and the ξ-rule (the congruence rule for λ). Since the axioms and rules defining weak equality are simply a subset of the rules defining $\beta\eta$-equality we adopt the convention that we write $\simeq$ if the rule applies to both, but use $\simeq_{\beta\eta\sigma}$ if it only applies to strong equality. Intuitively the weak equality captures the fragment where we never go under a λ.

Conversion for terms

First we show the rules for how terms interact with substitutions.

$$\begin{array}{llll}
\varnothing[\vec{t};t] & \simeq & t & \mathsf{proj} \\
t[\mathsf{id}_\Gamma] & \simeq & t & \mathsf{id} \\
t[\vec{t}\circ\vec{u}] & \simeq & t[\vec{t}][\vec{u}] & \mathsf{comp} \\
(\lambda_\sigma t)[\vec{t}] & \simeq_{\beta\eta\sigma} & \lambda_\sigma t[\vec{t}\circ{\uparrow_\sigma};\varnothing] & \mathsf{lam} \\
(t\,u)[\vec{t}] & \simeq & t[\vec{t}]\;u[\vec{t}] & \mathsf{capp}
\end{array}$$

Note that in the weak theory we are not allowed to push a substitution under λ, because otherwise we could derive ξ from the other congruences. Also, we do not have the β-rule. Instead we have a variant which we call $\beta\sigma$:

$$(\lambda_\sigma t)[\vec{u}]\;u \quad\simeq\quad t[\vec{u};u] \quad \beta\sigma$$

We also add the η-rule for the $\beta\eta$-equality:

$$t \quad\simeq\quad \lambda_\sigma(t[{\uparrow_\sigma}]\,\varnothing) \quad \eta$$

In addition we have refl, sym and trans and all congruence rules for terms except for ξ which only holds for the strong equality:

$$\frac{\begin{array}{c} t, u \in \mathsf{Tm}\,(\Gamma;\sigma)\,\tau \\ t \simeq_{\beta\eta\sigma} u\end{array}}{\lambda_\sigma t \simeq_{\beta\eta\sigma} \lambda_\sigma u}\;\xi$$

Conversion for substitutions

The conversion for substitutions is given by the usual laws defining a category:

$$\begin{array}{llll}
(\vec{t}\circ\vec{u})\circ\vec{v} & \simeq & \vec{t}\circ(\vec{u}\circ\vec{v}) & \mathsf{assoc} \\
\mathsf{id}_\Gamma\circ\vec{u} & \simeq & \vec{u} & \mathsf{idl} \\
\vec{u}\circ\mathsf{id}_\Gamma & \simeq & \vec{u} & \mathsf{idr}
\end{array}$$

and the following laws which formalize the existence of the appropriate finite products:

$$\begin{array}{lcll}
\uparrow_\sigma \circ (\vec{u}; u) & \simeq & \vec{u} & \mathsf{wk} \\
(\vec{t}; t) \circ \vec{u} & \simeq & (\vec{t} \circ \vec{u}); t[\vec{u}] & \mathsf{cons} \\
\mathsf{id}_{\Gamma;\sigma} & \simeq & (\mathsf{id}_\Gamma \circ \uparrow_\sigma); \varnothing & \mathsf{sid}
\end{array}$$

The choice of laws is motivated by the need to show soundness and completeness of our normalisation algorithm. In addition we have refl, sym and trans and all congruence rules for substitutions.

β and $\beta\sigma$

We note that the usual β-rule

$$(\lambda_\sigma t)\, u \quad \simeq_{\beta\eta\sigma} \quad t[u] \quad \beta$$

is too weak for the weak equality because we cannot reduce a λ-term with a delayed substitution.

We will show below that in the strong theory $\beta\sigma$ and β are equivalent.

Proposition 1
The rules β and $\beta\sigma$ are inter-derivable.

Proof
First of all it is easy to see that $\beta\sigma$ implies β: $(\lambda_\sigma t)\, u \simeq (\lambda_\sigma t[\mathsf{id}_\Gamma])\, u$ using id and $(\lambda_\sigma t[\mathsf{id}_\Gamma])\, u \simeq t[\mathsf{id}_\Gamma; u]$ using $\beta\sigma$. Secondly we show that the other direction is provable:

$$\begin{array}{cll}
 & ((\lambda_\sigma t)[\vec{u}])\, u & \\
\simeq & (\lambda_\sigma t[\vec{u} \circ \uparrow_\sigma; \varnothing])\, u & \{\mathsf{lam}\} \\
\simeq & t[\vec{u} \circ \uparrow_\sigma; \varnothing][\mathsf{id}_\Gamma; u] & \{\beta\} \\
\simeq & t[(\vec{u} \circ \uparrow_\sigma; \varnothing) \circ (\mathsf{id}_\Gamma; u)] & \{\mathsf{comp}\} \\
\simeq & t[(\vec{u} \circ \uparrow_\sigma) \circ (\mathsf{id}_\Gamma; u); \varnothing[\mathsf{id}_\Gamma; u]] & \{\mathsf{cons}\} \\
\simeq & t[(\vec{u} \circ \uparrow_\sigma) \circ (\mathsf{id}_\Gamma; u); u] & \{\mathsf{proj}\} \\
\simeq & t[\vec{u} \circ (\uparrow_\sigma \circ (\mathsf{id}_\Gamma; u)); u] & \{\mathsf{assoc}\} \\
\simeq & t[\vec{u} \circ \mathsf{id}_\Gamma; u] & \{\mathsf{wk}\} \\
\simeq & t[\vec{u}; u] & \{\mathsf{idr}\}
\end{array}$$

□

3 Recursive Normalisation

We start with a recursive implementation of normalisation and will later verify that it is terminating, sound and complete. However, since our implementation uses dependent types the function is automatically type-correct — we will never have to verify a property like subject reduction.

We start with an sketch of the top-level structure of the algorithm before going into the details of the implementation. We take the liberty of refering to values, environments and normal forms before defining them. Normalisation proceeds in two steps: we define a simple evaluator, basically an environment machine, which produces values, or weak normal forms:

$$\frac{t : \mathsf{Tm}\,\Delta\,\sigma \quad \vec{v} : \mathsf{Env}\,\Gamma\,\Delta}{\mathbf{eval}\,t\,\vec{v} : \mathsf{Val}\,\Gamma\,\sigma}$$

The evaluator is parameterized by an environment, which assigns to every free variable a value of the appropriate type and returns a value. To complete normalisation we define a quoting function which returns a normal form by recursively evaluating the term:

$$\frac{v : \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{quote}\,v : \mathsf{Nf}\,\Gamma\,\sigma}$$

Hence we obtain **nf** by combining **eval** and **quote**:

$$\frac{t : \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{nf}\,t : \mathsf{Nf}\,\Gamma\,\sigma} \quad \underline{\text{where}} \quad \mathbf{nf}\,t \Rightarrow \mathbf{quote}\,(\mathbf{eval}\;t\,\mathbf{id}_\Gamma)$$

here $\mathbf{id}_\Gamma$ is the identity environment, which we define by recursion over Γ. The definition will become clear when we have introduced some more types and operations.

$$\frac{\Gamma : \mathsf{Con}}{\mathbf{id}_\Gamma : \mathsf{Env}\,\Gamma\,\Gamma} \quad \underline{\text{where}}$$

$$\begin{array}{lll} \mathbf{id}_\varepsilon & \Rightarrow & \varepsilon \\ \mathbf{id}_{(\Gamma;\sigma)} & \Rightarrow & (\mathbf{id}_\Gamma)^{+}_{\sigma};\varnothing \end{array}$$

Having completed our sketch we start to fill in the details. We begin with the definition of de Bruijn variables — the variable $\varnothing$ refers is the variable at the (right-hand) end of the context. $\varnothing^{+}$is the next one in etc.

$$\frac{\Gamma : \mathsf{Con} \quad \sigma : \mathsf{Ty}}{\mathsf{Var}\,\Gamma\,\sigma : \star} \quad \underline{\text{where}} \quad \frac{}{\varnothing : \mathsf{Var}\,(\Gamma;\sigma)\,\sigma} \quad \frac{x : \mathsf{Var}\,\Gamma\,\sigma}{x^{+}_{\tau} : \mathsf{Var}\,(\Gamma;\tau)\,\sigma}$$

We define a type of neutral values, representing computations which are stuck due to the presence of variables in a key position. Since we need neutral values and neutral normal forms we parameterize the definition by an abstract type of values:

$$\frac{T : \mathsf{Con} \to \mathsf{Ty} \to \star \quad \Gamma : \mathsf{Con} \quad \sigma : \mathsf{Ty}}{\mathsf{Ne}^{T}\,\Gamma\,\sigma : \star} \quad \underline{\text{where}}$$

$$\frac{x : \mathsf{Var}\,\Gamma\,\sigma}{x : \mathsf{Ne}^{T}\,\Gamma\,\sigma} \quad \frac{f : \mathsf{Ne}^{T}\,\Gamma\,\sigma{\to}\tau \quad a : T\,\Gamma\,\sigma}{f\,a : \mathsf{Ne}^{T}\,\Gamma\,\tau}$$

Now a value is either a λ-closure or a neutral value. We also define the type of environments since it has to be defined mutually with the type of values:

$$\frac{\Gamma : \mathsf{Con} \quad \sigma : \mathsf{Ty}}{\mathsf{Val}\,\Gamma\,\sigma : \star} \quad \frac{\Gamma, \Delta : \mathsf{Con}}{\mathsf{Env}\,\Gamma\,\Delta : \star} \quad \underline{\text{where}}$$

$$\frac{t : \mathsf{Tm}\,(\Delta;\sigma)\,\tau \quad \vec{v} : \mathsf{Env}\,\Gamma\,\Delta}{\lambda_\sigma t[\vec{v}] : \mathsf{Val}\,\Gamma\,(\sigma\to\tau)} \quad \frac{n : \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma}{n : \mathsf{Val}\,\Gamma\,\sigma}$$

$$\frac{}{\varepsilon : \mathsf{Env}\,\Gamma\,\varepsilon} \quad \frac{v : \mathsf{Val}\,\Gamma\,\sigma \quad \vec{v} : \mathsf{Env}\,\Gamma\,\Delta}{(\vec{v}; v) : \mathsf{Env}\,\Gamma\,(\Delta;\sigma)}$$

We are ready to define evaluation which has to be defined mutually with evaluation of substitutions and applications of values:

$$\frac{t : \mathsf{Tm}\,\Delta\,\sigma \quad \vec{v} : \mathsf{Env}\,\Gamma\,\Delta}{\mathbf{eval}\,t\,\vec{v} : \mathsf{Val}\,\Gamma\,\sigma} \quad \frac{\vec{t} : \mathsf{Subst}\;\Gamma\,\Delta \quad \vec{v} : \mathsf{Env}\,B\,\Gamma}{\vec{\mathbf{eval}}\,\vec{t}\,\vec{v} : \mathsf{Env}\,B\,\Delta}$$

$$\frac{f : \mathsf{Val}\,\Gamma\,(\sigma\to\tau) \quad a : \mathsf{Val}\,\Gamma\,\sigma}{f \mathbin{@} a : \mathsf{Val}\,\Gamma\,\tau}$$

The definition of **eval** is the straightforward implementation of an environment based evaluator:

$$\begin{array}{llll}
\mathbf{eval} & \varnothing & (\vec{v}; v) & \Rightarrow & v \\
\mathbf{eval} & t[\vec{t}] & \vec{v} & \Rightarrow & \mathbf{eval}\,t\,(\vec{\mathbf{eval}}\,\vec{t}\,\vec{v}) \\
\mathbf{eval} & \lambda t & \vec{v} & \Rightarrow & \lambda t[\vec{v}] \\
\mathbf{eval} & t\,u & \vec{v} & \Rightarrow & (\mathbf{eval}\,t\,\vec{v}) \mathbin{@} (\mathbf{eval}\,u\,\vec{v})
\end{array}$$

We also have to evaluate substitutions:

$$\begin{array}{lllll}
\vec{\mathbf{eval}} & \mathsf{id} & \vec{v} & \Rightarrow & \vec{v} \\
\vec{\mathbf{eval}} & \vec{t} \circ \vec{u} & \vec{v} & \Rightarrow & \vec{\mathbf{eval}}\,\vec{t}\,(\vec{\mathbf{eval}}\,\vec{u}\,\vec{v}) \\
\vec{\mathbf{eval}} & (\vec{t}; t) & \vec{v} & \Rightarrow & (\vec{\mathbf{eval}}\,\vec{t}\,\vec{v}); (\mathbf{eval}\,t\,\vec{v}) \\
\vec{\mathbf{eval}} & \uparrow_\sigma & (\vec{v}; v) & \Rightarrow & \vec{v}
\end{array}$$

Application of values recursively calls **eval** on the term with the context extended by the argument, while in the case of a neutral value n, the argument is added to the spine:

$$\begin{array}{lllll}
\lambda t[\vec{v}] & @ & a & \Rightarrow & \mathbf{eval}\,t\,(\vec{v}; a) \\
n & @ & a & \Rightarrow & n\,a
\end{array}$$

To define full normalisation we first define so called η-long β-normal forms, reusing the definition of neutral values:

$$\frac{\Gamma : \mathsf{Con} \quad \sigma : \mathsf{Ty}}{\mathsf{Nf}\,\Gamma\,\sigma : \star} \quad \underline{\text{where}} \quad \frac{n : \mathsf{Nf}\,(\Gamma;\sigma)\,\tau}{\lambda_\sigma n : \mathsf{Nf}\,\Gamma\,(\sigma\to\tau)} \quad \frac{n : \mathsf{Ne}^{\mathsf{Nf}}\,\Gamma\,\bullet}{n : \mathsf{Nf}\,\Gamma\,\bullet}$$

Alternatively, β normal forms can be defined by allowing any type in the last rule, instead of restricting it to base type.

Weakening is defined by mutual recursion for neutral values, values, environments.

$$\frac{v \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma}{v^{+}_{\tau} \;:\; \mathsf{Ne}^{\mathsf{Val}}\,(\Gamma;\tau)\,\sigma} \quad \underline{\text{where}} \quad \begin{array}{lcl} x^{+}_{\sigma} & \Rightarrow & x^{+}_{\sigma} \\ n\,v^{+}_{\sigma} & \Rightarrow & (n^{+}_{\sigma})\,@\,(v^{+}_{\sigma}) \end{array}$$

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{v^{+}_{\tau} \;:\; \mathsf{Val}\,(\Gamma;\tau)\,\sigma} \quad \underline{\text{where}} \quad \begin{array}{lcl} n^{+}_{\sigma} & \Rightarrow & n^{+}_{\sigma} \\ (\lambda_{\sigma}\,t[\vec{v}])^{+}_{\sigma} & \Rightarrow & \lambda_{\sigma}\,t[\vec{v}^{+}_{\sigma}] \end{array}$$

$$\frac{v \;:\; \mathsf{Env}\,\Gamma\,\Delta}{v^{+}_{\sigma} \;:\; \mathsf{Env}\,(\Gamma;\sigma)\,\Delta} \quad \underline{\text{where}} \quad \begin{array}{lcl} \varepsilon^{+}_{\sigma} & \Rightarrow & \varepsilon \\ (\vec{v};v)^{+}_{\sigma} & \Rightarrow & (\vec{v}^{+}_{\sigma});(v^{+}_{\sigma}) \end{array}$$

We can iterate weakenings using contexts, here $+\!\!+$ is concatenation of contexts. We give the instance for values as an example:

$$\frac{\Delta \;:\; \mathsf{Con} \quad v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{v^{+}_{\Delta} \;:\; \mathsf{Val}\,(\Gamma +\!\!+ \Delta)\,\sigma} \quad \underline{\text{where}} \quad \begin{array}{lcl} v^{+}_{\varepsilon} & \Rightarrow & v \\ v^{+}_{(\Delta;\,\sigma)} & \Rightarrow & (v^{+}_{\Delta})^{+}_{\sigma} \end{array}$$

The same principle applies to all weakening operations.

We introduce a family of (overloaded) embedding operations $\ulcorner\_\urcorner$ for each of the following types, and their composites:

$$\begin{array}{lcl} \mathsf{Var}\,\Gamma\,\sigma & \hookrightarrow & \mathsf{Tm}\,\Gamma\,\sigma \\ \mathsf{Val}\,\Gamma\,\sigma & \hookrightarrow & \mathsf{Tm}\,\Gamma\,\sigma \\ \mathsf{Nf}\,\Gamma\,\sigma & \hookrightarrow & \mathsf{Tm}\,\Gamma\,\sigma \\ \mathsf{Env}\,\Gamma\,\Delta & \hookrightarrow & \mathsf{Subst}\,\Gamma\,\Delta \end{array}$$

These are easy to define. We are ready to define **quote** for values simultaneously with $\overline{\mathbf{quote}}$ for neutral values:

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{quote}_{\sigma}\,v \;:\; \mathsf{Nf}\,\Gamma\,\sigma} \quad \frac{n \;:\; \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma}{\overline{\mathbf{quote}}\,n \;:\; \mathsf{Ne}^{\mathsf{Nf}}\,\Gamma\,\sigma} \quad \underline{\text{where}}$$

$$\begin{array}{llcl} \mathbf{quote}_{\bullet} & n & \Rightarrow & \overline{\mathbf{quote}}\,n \\ \mathbf{quote}_{(\sigma\to\tau)} & f & \Rightarrow & \lambda_{\sigma}\mathbf{quote}_{\tau}\,(f^{+}_{\sigma}\,@\,\varnothing) \end{array}$$

$$\begin{array}{llcl} \overline{\mathbf{quote}} & x & \Rightarrow & x \\ \overline{\mathbf{quote}} & n\,v & \Rightarrow & (\overline{\mathbf{quote}}\,n)\,(\mathbf{quote}\,v) \end{array}$$

Note that we define **quote** by recursion over the type. This can be avoided, if we are only interested in β-normal forms. In this case we would define **quote** as follows:

$$\begin{array}{llcl} \mathbf{quote}^{\beta} & \lambda_{\sigma}\,t[\vec{v}] & \Rightarrow & \lambda_{\sigma}\mathbf{quote}^{\beta}_{\tau}\,(\mathbf{eval}\,t\,(\vec{v}^{+}_{\sigma};\varnothing)) \\ \mathbf{quote}^{\beta} & n & \Rightarrow & \overline{\mathbf{quote}}^{\beta}\,n \end{array}$$

While **quote** and $\overline{\mathbf{quote}}$ remind us on reify and reflect (sometimes also called quote and unquote) as they appear in Normalisation by Evaluation, the precise relation is less clear: while reify maps semantical values to normal forms and reflect maps neutral terms to semantic values, here both **quote** and $\overline{\mathbf{quote}}$ go basically in the same direction mapping computational values resp. neutral values to normal forms, resp. normal forms.

4 Big-step semantics

The functions defined in the previous section are not structurally recursive and hence it is not obvious how to implement them in total Type Theory. To bridge this gap we will exploit a technique pioneered in (Bove & Capretta, 2001): we inductively define the graph of our function and then show that the graph is total: i.e. for every input there exists an output. We can use this proof to actually run our function without having to employ a choice principle — i.e. we keep the separation of propositions and types.

The Bove-Capretta technique

As an example consider the following recursive function which is defined using nested recursion:

$$\frac{n \;:\; \mathsf{Nat}}{\mathbf{f}\; n \;:\; \mathsf{Nat}} \quad \underline{\text{where}} \quad \begin{array}{llll} \mathbf{f} & \mathsf{zero} & \Rightarrow & \mathsf{zero} \\ \mathbf{f} & (\mathsf{suc}\; n) & \Rightarrow & \mathbf{f}\,(\mathbf{f}\; n) \end{array}$$

While it is obvious to us that $\mathbf{f}$ is total, it is not obviously structural recursive. However, we can inductively define the graph of the function as a relation — it's big-step semantics:

$$\frac{n,\; n' \;:\; \mathsf{Nat}}{\mathsf{f}\; n \Downarrow n' \;:\; \mathsf{Prop}} \quad \underline{\text{where}} \quad \frac{}{\mathsf{fz} \;:\; \mathsf{f}\,\mathsf{zero} \Downarrow \mathsf{zero}} \quad \frac{p \;:\; \mathsf{f}\; n \Downarrow n' \quad p' \;:\; \mathsf{f}\; n' \Downarrow n''}{\mathsf{fs}\; p\; p' \;:\; \mathsf{f}\,\mathsf{suc}\; n \Downarrow n''}$$

We adopt the convention that the relation corresponding to the recursive definition of $\mathbf{f} \;:\; \mathsf{Nat} \to \mathsf{Nat}$ is written as $\mathsf{f} - \Downarrow - \;:\; \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Prop}$ [7] . We can now define a structurally structurally recursively version of $\mathbf{f}$ called $\mathbf{f}^{\mathbf{str}}$

$$\frac{p \;:\; \mathsf{f}\; n \Downarrow n'}{\mathbf{f}^{\mathbf{str}}\; n\; p \;:\; \Sigma\, n'\!:\!\mathsf{Nat}\,.\, n' = n} \quad \underline{\text{where}}$$

$$\begin{array}{llllll} \mathbf{f}^{\mathbf{str}} & \mathsf{zero} & \mathsf{fz} & \Rightarrow & (\mathsf{zero},\, \mathsf{refl}) & \\ \mathbf{f}^{\mathbf{str}} & (\mathsf{suc}\; n) & (\mathsf{fs}\; p\; p') & \underline{\text{with}} & \mathbf{f}^{\mathbf{str}}\; n\; p & \\ & & & \mid & (n',\, \mathsf{refl}) & \Rightarrow \quad \mathbf{f}^{\mathbf{str}}\; n'\; p' \end{array}$$

And once we have established that $\mathsf{f} - \Downarrow -$ is total:

Theorem 2

$$\frac{n \;:\; \mathsf{Nat}}{\mathsf{f}\; n \Downarrow \mathsf{zero}}$$

Proof
By induction on n. □

We now can redefine $\mathbf{f}$ as a structurally recursive function: [8]

$$\frac{n \;:\; \mathsf{Nat}}{\mathbf{f}\; n \;:\; \mathsf{Nat}} \quad \mathbf{f} \quad n \quad \Rightarrow \quad (\mathbf{fst}\,(\mathbf{f}^{\mathbf{str}}\; n\,(\mathbf{theorem2}\; n)))$$

[7] For those watching in colour note the associated colour difference.

[8] We use the notation that theorems given in this paper can be referred to by name in other definitions by the name $\mathbf{theorem}n$ where n is the number of the theorem.

Big-step semantics of nf

We will now apply this technique to the recursive definition of normalisation from the previous section. The big-step semantics is given by the following inductively defined relations:

$$\frac{t\ :\ \mathsf{Tm}\,\Delta\,\sigma \quad \vec{v}\ :\ \mathsf{Env}\,\Gamma\,\Delta \quad v\ :\ \mathsf{Val}\,\Gamma\,\sigma}{\mathsf{eval}\,t\,\vec{v}\ \Downarrow\ v\ :\ \mathsf{Prop}}$$

$$\frac{\vec{t}\ :\ \mathsf{Subst}\,\Gamma\,\Delta \quad \vec{v}\ :\ \mathsf{Env}\,B\,\Gamma \quad \vec{w}\ :\ \mathsf{Env}\,B\,\Delta}{\overrightarrow{\mathsf{eval}}\,\vec{t}\,\vec{v}\ \Downarrow\ \vec{w}\ :\ \mathsf{Prop}}$$

$$\frac{f\ :\ \mathsf{Val}\,\Gamma\,(\sigma{\to}\tau) \quad a\ :\ \mathsf{Val}\,\Gamma\,\sigma \quad v\ :\ \mathsf{Val}\,\Gamma\,\tau}{f\ @\ a \Downarrow v\ :\ \mathsf{Prop}}$$

$$\frac{v\ :\ \mathsf{Val}\,\Gamma\,\sigma \quad n\ :\ \mathsf{Nf}\,\Gamma\,\sigma}{\mathsf{quote}\,v\ \Downarrow\ n\ :\ \mathsf{Prop}} \qquad \frac{v\ :\ \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma \quad n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\,\Gamma\,\sigma}{\overline{\mathsf{quote}}\,v\ \Downarrow\ n\ :\ \mathsf{Prop}}$$

$$\frac{t\ :\ \mathsf{Tm}\,\Gamma\,\sigma \quad n\ :\ \mathsf{Nf}\,\Gamma\,\sigma}{\mathsf{nf}\,t\ \Downarrow\ n\ :\ \mathsf{Prop}}$$

The inductive definition of those relations is straightforward from the recursive definition of the functions in the previous section. To illustrate this we give the constructors for $\mathsf{eval}\,t\,\vec{v}\ \Downarrow\ v$:

$$\frac{}{\mathsf{rlam}\ :\ \mathsf{eval}\,(\lambda_\sigma t)\,\vec{v}\ \Downarrow\ \lambda_\sigma t[\vec{v}]} \qquad \frac{}{\mathsf{rvar}\ :\ \mathsf{eval}\,\varnothing\,(\vec{v}; v)\ \Downarrow\ v}$$

$$\frac{p\ :\ \overrightarrow{\mathsf{eval}}\,\vec{t}\,\vec{v}\ \Downarrow\ \vec{v}' \quad q\ :\ \mathsf{eval}\,t\,\vec{v}'\ \Downarrow\ v}{\mathsf{rsubs}\,p\,q\ :\ \mathsf{eval}\,(t[\vec{t}])\,\vec{v}\ \Downarrow\ v}$$

$$\frac{p\ :\ \mathsf{eval}\,t\,\vec{v}\ \Downarrow f \quad q\ :\ \mathsf{eval}\,u\,\vec{v}\ \Downarrow\ v \quad r\ :\ f\ @\ v \Downarrow\ w}{\mathsf{rapp}\,p\,q\,r\ :\ \mathsf{eval}\,(t\,u)\,\vec{v}\ \Downarrow\ w}$$

We can now augment our evaluation algorithm, making it structurally recursive on the big-step relation. To make the induction go through we have to show simultaneously that the functions calculate the specified results. We mutually define structural recursive functions corresponding to the recursive ones in the previous section by structural recursion over the proofs of termination, e.g. in the case of

eval and **nf**: [9]

$$\frac{t \;:\; \mathsf{Tm}\,\Delta\,\sigma \qquad \vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta \qquad p \;:\; \mathsf{eval}\,t\,\vec{v} \Downarrow v}{\mathbf{eval}^{\mathbf{str}}\,t\,\vec{v}\,p \;:\; \Sigma v'{:}\mathsf{Val}\,\Gamma\,\sigma\,.\,v' = v} \quad \underline{\text{where}}$$

$$\begin{array}{llllllllll}
\mathbf{eval}^{\mathbf{str}} & \varnothing & (\vec{v};v) & \mathsf{rvar} & \Rightarrow & (v,\,\mathsf{refl}) & & & & \\
\mathbf{eval}^{\mathbf{str}} & t[\vec{t}\,] & \vec{v} & (\mathsf{rsubs}\,p\,q) & \underline{\text{with}} & \overrightarrow{\mathbf{eval}}^{\mathbf{str}}\,\vec{t}\,\vec{v}\,p & & & & \\
 & & & & \mid & (\vec{v}',\,\mathsf{refl}) & \Rightarrow & \mathbf{eval}^{\mathbf{str}}\,t\,\vec{v}'\,q & & \\
\mathbf{eval}^{\mathbf{str}} & \lambda t & \vec{v} & \mathsf{rlam} & \Rightarrow & (\lambda\,t[\vec{v}],\,\mathsf{refl}) & & & & \\
\mathbf{eval}^{\mathbf{str}} & t\,u & \vec{v} & (\mathsf{rapp}\,p\,q\,r) & \underline{\text{with}} & \mathbf{eval}^{\mathbf{str}}\,t\,\vec{v}\,p & \mid & \mathbf{eval}^{\mathbf{str}}\,u\,\vec{v}\,q & & \\
 & & & & \mid & (f,\,\mathsf{refl}) & \mid & (a,\,\mathsf{refl}) & \Rightarrow & f \mathbin{@^{\mathbf{str}}_{r}} a
\end{array}$$

$$\frac{p \;:\; \mathsf{nf}\,t \Downarrow n}{\mathbf{nf}^{\mathbf{str}}\,t\,p \;:\; \mathsf{Nf}\,\Gamma\,\sigma} \quad \underline{\text{where}}$$

$$\begin{array}{llllll}
\mathbf{nf}^{\mathbf{str}} & t & (\mathsf{rnf}\,p\,p') & \underline{\text{with}} & \mathbf{eval}^{\mathbf{str}}\,t\,p & \\
 & & & \mid & (v,\,\mathsf{refl}) & \Rightarrow \quad \mathbf{quote}^{\mathbf{str}}\,v\,p'
\end{array}$$

We are using the <u>with</u>-construct here to allow us to pattern match on an intermediate value - see (McBride & McKinna, 2004; Norell, 2007b) for further details. The derivation of structurally recursive versions of $@$, $\overrightarrow{\mathbf{eval}}$, **quote**, $\overline{\mathbf{quote}}$ proceeds analogously.

5 Termination and Completeness

We use the notion of strong computability to show that our normalisation function terminates and that the result is $\beta\eta$-equivalent to the input. Since we are evaluating under λ, we introduce a Kripke-style extension of computability at higher type.

$$\frac{v \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathsf{SCV}_{\Gamma,\sigma}\,v \;:\; \mathsf{Prop}}$$

which is defined by recursion over σ: [10]

$$\frac{\overline{\mathsf{quote}}\,n \Downarrow m \qquad \ulcorner n \urcorner \simeq \ulcorner m \urcorner}{\mathsf{SCV}_{\Gamma,\bullet}\,n}$$

$$\frac{\forall\Delta.\forall v{:}\mathsf{Val}\,(\Gamma +\!\!\!+ \Delta)\,\sigma\,.\,\mathsf{SCV}\,v \rightarrow \exists w.f^{+}_{\Delta} \mathbin{@} v \Downarrow w \wedge \ulcorner f^{+}_{\Delta} \urcorner \ulcorner v \urcorner \simeq \ulcorner w \urcorner \wedge \mathsf{SCV}\,w}{\mathsf{SCV}_{\Gamma,\,(\sigma\rightarrow\tau)}\,f}$$

It is straightforward to extend strong computability to environments:

$$\frac{\vec{v} \;:\; \mathsf{Env}\,\Gamma\,\Delta}{\mathsf{SCE}_{\Gamma,\,\Delta}\,\vec{v} \;:\; \mathsf{Prop}} \quad \underline{\text{where}} \quad \frac{}{\mathsf{SCE}\,\varepsilon} \qquad \frac{\mathsf{SCE}\,\vec{v} \qquad \mathsf{SCV}\,v}{\mathsf{SCE}\,(\vec{v};v)}$$

We will need that strong computability is closed under weakening:

[9] Note, however that we never use the proof to make a choice.

[10] We find it convenient to use the same notation as for inductive definitions. However this is not strictly positive so in the formalisation we use recursive definitions.

Lemma 3

$$\frac{\mathsf{SCV}_{\Gamma,\,\sigma}\ v}{\mathsf{SCV}_{(\Gamma +\!\!+ \Delta),\,\sigma}\ v^{+}_{\Delta}} \qquad \frac{\mathsf{SCE}_{B,\,\Delta}\ \vec{v}}{\mathsf{SCE}_{(B +\!\!+ \Gamma),\,\Delta}\ \vec{v}^{+}_{\Gamma}}$$

Proof
By induction over σ and Σ. □

Our main technical lemma is that **quote** terminates for all strongly computable values and that the result is $\beta\eta\sigma$-convertible to the input. Our proof proceeds by induction over the type, to deal with the negative occurrence of types we show at the same time that termination of quote for neutral terms implies strong computability. The second component of our proof is also required to show that the identity environment is strongly computable. This structure of establishing two propositions by mutual induction over types is common to conventional strong normalisation proofs and can also be found in the normalisation by evaluation construction.

Lemma 4

$$\frac{\mathsf{SCV}_{\Gamma,\,\sigma}\ v}{\mathsf{quote}_{\Gamma,\,\sigma}\ v {\Downarrow} m \wedge \ulcorner v \urcorner \simeq \ulcorner m \urcorner}(q) \qquad \frac{\overline{\mathsf{quote}}_{\Gamma,\,\sigma}\ n {\Downarrow} m \quad \ulcorner n \urcorner \simeq \ulcorner m \urcorner}{\mathsf{SCV}_{\Gamma,\,\sigma}\ n}(u)$$

Proof
By mutual induction over σ. In the base case both implications follow trivially from the definition of SCV and the observation that all values of base type are neutral. Consider $(\sigma{\to}\tau)$:

(q) Given $\mathsf{SCV}_{\Gamma,\,\sigma}\, f$. Using ind.hyp. (u) for σ we can show that $\mathsf{SCV}_{(\Gamma;\,\sigma),\,\sigma}\, \varnothing$, and hence $f^{+}_{\sigma}\ @\ \varnothing \Downarrow v$ (1), $\ulcorner f^{+}_{\sigma} \urcorner \varnothing \simeq_{\mathsf{w}\sigma} \ulcorner v \urcorner$ (2) and $\mathsf{SCV}_{\Gamma;\,\sigma,\tau}\, v$. Now using ind.hyp. (q) for τ we know that $\mathsf{quote}\, v \Downarrow n$ (3) and $\ulcorner v \urcorner {\simeq} \ulcorner n \urcorner$ (4). By the definition of the big-step semantics and (1,2) we can infer that $\mathsf{quote}_{\Gamma,\,\sigma}\, f \Downarrow \lambda_{\sigma} n$ and using η, ξ and with (2,4) that $\ulcorner f \urcorner {\simeq} \ulcorner \lambda_{\sigma} n \urcorner$.

(u) Given $\overline{\mathsf{quote}}_{\Gamma,\,(\sigma\to\tau)}\, n \Downarrow m$ and $\ulcorner n \urcorner \simeq \ulcorner m \urcorner$ (1). To show $\mathsf{SCV}_{\Gamma,\sigma\to\tau}\, n$, assume as given $\mathsf{SCV}_{(\Gamma +\!\!+ \Delta),\,\sigma}\, v$. Certainly $n^{+}_{\Delta}\ @\ v \Downarrow n^{+}_{\Delta}\, v$ since n is neutral. By ind.hyp. (q) for σ we know that $\mathsf{quote}_{\Gamma,\sigma}\, v \Downarrow u$ and $\ulcorner v \urcorner \simeq \ulcorner u \urcorner$ (2). Hence, $\overline{\mathsf{quote}}_{\Gamma,\sigma}\, (n^{+}_{\Delta}\, v) \Downarrow m\, u$ (3) and from (1,2) we can infer $\ulcorner n^{+}_{\Delta} \urcorner \ulcorner v \urcorner {\simeq} \ulcorner n^{+}_{\Delta}\, v \urcorner$ (4). $\mathsf{SCV}_{(\Gamma +\!\!+ \Delta),\,\tau}\, (n^{+}_{\Delta}\, v)$ follows from (3) and (4) by ind. hyp. (u) for τ.

□

A simple consequence of the 2nd component of the lemma is that variables are strongly computable and hence the identity environment is strongly computable.

Corollary 5

$$\frac{x\ :\ \mathsf{Var}\ \Gamma\ \sigma}{\mathsf{SCV}\ x}(1) \qquad \frac{\Gamma\ :\ \mathsf{Con}}{\mathsf{SCE}\ \mathbf{id}_{\Gamma}}(2)$$

We prove the fundamental theorem for our notion of strong computability which has to be shown mutually for terms and substitutions:

Theorem 6

$$\frac{t\ :\ \mathsf{Tm}\,\Delta\,\sigma \quad \mathsf{SCE}_{\Gamma,\Delta}\,\vec{v}}{\exists v : \mathsf{Val}\,\Gamma\,\sigma \wedge \mathsf{eval}\,t\,\vec{v} \Downarrow v \wedge t[\ulcorner\vec{v}\urcorner] \simeq_{\mathsf{w}\sigma} \ulcorner v \urcorner \wedge \mathsf{SCV}\,v}$$

$$\frac{\vec{t}\ :\ \mathsf{Subst}\,\Gamma\,\Delta \quad \mathsf{SCE}_{B,\,\Gamma}\,\vec{v}}{\exists\,\vec{w}\ :\ \mathsf{Env}\,B\,\Delta\,.\,\overrightarrow{\mathsf{eval}}\,\vec{t}\,\vec{v} \Downarrow \vec{w} \wedge \vec{t} \circ \ulcorner\vec{v}\urcorner \simeq_{\mathsf{w}\sigma} \ulcorner\vec{w}\urcorner \wedge \mathsf{SCE}\,\vec{w}}$$

Proof

By induction over $t\ :\ \mathsf{Tm}\,\Delta\,\sigma$ and $\vec{t}\ :\ \mathsf{Subst}\,\Gamma\,\Delta$ using the laws of the weak conversion relation and the definition of the big-step reduction relation. The proof is mostly straightforward adaptation of the fundamental theorem for logical predicates we just discuss some interesting cases. We assume as given $\mathsf{SCE}_{B,\,\Gamma}\,\vec{v}$.

$\lambda_\sigma t$**:** Since $\lambda_\sigma t$ is a value, we have that $\mathsf{eval}\,(\lambda_\sigma t)\,\vec{v} \Downarrow f$ with $f = \lambda_\sigma t[\vec{v}]$ and the equational condition holds trivially. It remains to show that $\mathsf{SCV}_{\Gamma,\sigma\to\tau}\,f$. Assume as given $\mathsf{SCV}_{\Gamma +\!\!+ \Delta,\sigma}\,v$, using the induction hypothesis for t and lemma 3 for $\vec{v}$, we know that there is a w, such that $\mathsf{eval}\,t\,(\vec{v}^{+}_{\Delta}; v) \Downarrow w$, $\ulcorner t[\vec{v}^{+}_{\Delta}; v]\urcorner \simeq_{\mathsf{w}\sigma} \ulcorner w \urcorner$ and $\mathsf{SCV}\,w$. Using our the definition of the big-step relation we have that $f^{+}_{\Delta}\ @\ v \Downarrow w$ and using $\beta\sigma$ we can show that $\ulcorner f \urcorner \ulcorner v \urcorner \simeq_{\mathsf{w}\sigma} \ulcorner w \urcorner$.

$(t\,u)$**:** By ind.hyp for t and u we can infer $\mathsf{eval}\,t\,\vec{v} \Downarrow f$ (1), $t[\ulcorner\vec{v}\urcorner] \simeq_{\mathsf{w}\sigma} \ulcorner f \urcorner$ (2) and $\mathsf{SCV}_{\Gamma,\,(\sigma\to\tau)}\,f$ (3); $\mathsf{eval}\,u\,\vec{v} \Downarrow v$ (4), $u[\ulcorner\vec{v}\urcorner] \simeq_{\mathsf{w}\sigma} \ulcorner v \urcorner$ (5) and $\mathsf{SCV}_{\Gamma,\,\sigma}\,v$ (6). By the definition of SCV using $\Delta = \varepsilon$ and (3,6) we get that $f\ @\ v \Downarrow w$ (7), $\ulcorner f \urcorner \ulcorner v \urcorner \simeq_{\mathsf{w}\sigma} \ulcorner w \urcorner$ (8) and $\mathsf{SCV}\,w$. Using the definition of the big-step semantics and (1,4,7) we can show that $\mathsf{eval}\,(t\,u)\,\vec{v} \Downarrow w$ and $(t\,u)[\ulcorner\vec{v}\urcorner] \simeq \ulcorner w \urcorner$ using capp and (2,5,8).

$(\vec{t}; t)$**:** By ind.hyp. for $\vec{t}$ we get $\overrightarrow{\mathsf{eval}}\,\vec{t}\,\vec{v} \Downarrow \vec{w}$ (1), $\vec{t} \circ \ulcorner\vec{v}\urcorner \simeq_{\mathsf{w}\sigma} \ulcorner\vec{w}\urcorner$ (2) and $\mathsf{SCE}\,\vec{w}$ (3). Using the latter with the ind.hyp. for t we have that $\overrightarrow{\mathsf{eval}}\,t\,\vec{v} \Downarrow v$ (4), $t[\ulcorner\vec{v}\urcorner] \simeq_{\mathsf{w}\sigma} \ulcorner v \urcorner$ (5) and $\mathsf{SCV}\,v$ (6). The definition of the big-step reduction and (1,4) imply that $\overrightarrow{\mathsf{eval}}\,(\vec{t}; t)\,\vec{v} \Downarrow (\vec{w}; v)$. Using cons and (2,5) we can show $(\vec{t}; t) \circ \ulcorner\vec{v}\urcorner \simeq_{\mathsf{w}\sigma} \ulcorner\vec{w}; v\urcorner$ and $\mathsf{SCE}\,(\vec{w},\ v)$ by (3,6).

□

Note that the proof never refers to the notion of computability at base type, hence we could have replaced it with any predicate[11] . The fundamental theorem already implies termination and completeness for reduction to values — this corresponds to our result in our workshop paper (Altenkirch & Chapman, 2006) which uses combinatory logic corresponding to weak equality of closed terms. Correspondingly we can actually show that the result is weakly equal ($\simeq_{\mathsf{w}\sigma}$) to its input, even though here we only need that it is $\beta\eta\sigma$-equal to its input.

We now can combine the results to infer that **nf** terminates and produces a normal form which is $\beta\eta\sigma$-equivalent to its input.

[11] Including the empty set, indeed they are no closed values of base type.

Proposition 7

$$\frac{t \ :\ \mathsf{Tm}\,\Delta\,\sigma}{\mathsf{nf}\ t {\Downarrow} n \wedge t\ \simeq\ \ulcorner n \urcorner}$$

Proof
By the fundamental theorem 6 and corollary 5(2) we know that $\mathsf{eval}\,t\,\mathbf{id}\ \Downarrow\ v$ with $t {\simeq_{\mathsf{w}\sigma}} t[\ulcorner \mathbf{id} \urcorner] \simeq_{\mathsf{w}\sigma} \ulcorner v \urcorner$ and $\mathsf{SCV}\ v$. Using lemma 4 we know that $\mathbf{quote}\,v {\Downarrow} n$ and $\ulcorner v \urcorner {\simeq} \ulcorner n \urcorner$ and hence by combining the two steps we obtain the result. □

Since we now know that our functions terminate, we can from now on use the total functions defined in section 4 together with the termination proofs given in this section.

$$\frac{t \ :\ \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{nf}\,t\ :\ \mathsf{Nf}\,\Gamma\,\sigma} \quad \underline{\text{where}} \quad \mathbf{nf}\,t\ \Rightarrow\ \mathbf{nf}^{\mathbf{str}}\ t\,(\mathbf{fst}\,(\mathbf{prop7}\ t))$$

To ease notation we will omit the proof terms altogether but make sure that we only use strongly computable values and environments. We show *stability* by simultaneous induction on normal and neutral terms

Proposition 8

$$\frac{n\ :\ \mathsf{Nf}\,\Gamma\,\sigma}{\mathbf{nf}\,n\ =\ n} \qquad \frac{n\ :\ \mathsf{Ne}^{\mathsf{Nf}}\,\Gamma\,\sigma}{\Sigma\, n' {:} \mathsf{Ne}^{\mathsf{Val}}\,\Gamma\,\sigma\,.\,\mathbf{eval}\,n = n'\ \wedge\ \overline{\mathbf{quote}}\,n' = n}$$

6 Soundness

It remains to be shown that normalisation maps $\beta\eta\sigma$-equivalent terms to equal normal forms. We define a logical relation on values which is preserved by the values obtained from convertible terms and which is mapped to identical normal forms by quote.

$$\frac{v, w\ :\ \mathsf{Val}\,\Gamma\,\sigma}{v \sim_{\Gamma,\,\sigma} w\ :\ \mathsf{Prop}} \quad \underline{\text{where}}$$

$$\frac{\overline{\mathbf{quote}}\,m = \overline{\mathbf{quote}}\,n}{m \sim_{\Gamma,\,\bullet} n}$$

$$\frac{\forall \Delta\,.\,\forall v, w{:}\mathsf{Val}\,(\Gamma {+\!\!+} \Delta)\,\sigma\,.\,v \sim w\ \rightarrow\ f_{\overset{+}{\Delta}} @ v \sim g_{\overset{+}{\Delta}} @ w}{f \sim_{\Gamma,\,(\sigma \rightarrow \tau)} g}$$

The pointwise extension to environments is straightforward:

$$\frac{\vec{v},\ \vec{w}\ :\ \mathsf{Env}\,\Gamma\,\Delta}{\vec{v} \sim \vec{w}\ :\ \mathsf{Prop}} \quad \underline{\text{where}} \quad \frac{}{\varepsilon \sim \varepsilon} \qquad \frac{\vec{v} \sim \vec{w} \quad v \sim w}{(\vec{v}; v) \sim (\vec{w}; w)}$$

As before for strong computability we will need that $\sim$ is closed under weakening:

Lemma 9

$$\frac{v \sim_{\Gamma,\sigma} w}{v_{\overset{+}{\Delta}} \sim_{(\Gamma {+\!\!+} \Delta),\sigma} w_{\overset{+}{\Delta}}} \qquad \frac{\vec{v} \sim_{\Gamma,\Sigma} \vec{w}}{v_{\overset{+}{\Delta}} \sim_{(\Gamma {+\!\!+} \Delta),\Sigma} w_{\overset{+}{\Delta}}}$$

Proof
By induction over σ and Σ. □

We will also need that we have defined a family of partial equivalence relations (PERs);

Lemma 10
For all $v,\ v' \ :\ \mathsf{Val}\,\Gamma\,\sigma$ such that $v \sim_{\Gamma,\sigma} v'$ is symmetric and transitive and for all $\vec{v},\ \vec{v}' \ :\ \mathsf{Env}\,\Gamma\,\Delta$ such that $\vec{v} \sim_{\Gamma,\Delta} \vec{v}'$ is symmetric and transitive.

Proof
By induction over σ for both properties for the value relation and corresponding by induction over Δ for the environment relation. Symmetry for environments requires symmetry for values and transitivity for environments requires transitivity for values. Note also that we need symmetry of values to establish transitivity of values for the $\sigma{\rightarrow}\tau$ case. □

Before we can establish the fundamental theorem for logical relations we have to show an *identity extension lemma*:

Lemma 11

$$\frac{t\ :\ \mathsf{Tm}\,\Gamma\,\sigma \qquad \vec{v} \sim \vec{w}}{\mathbf{eval}\,t\,\vec{v} \sim \mathbf{eval}\,t\,\vec{w}} \qquad \frac{\vec{t}\ :\ \mathsf{Subst}\,\Gamma\,\Delta \qquad \vec{v} \sim \vec{w}}{\vec{\mathbf{eval}}\,\vec{t}\,\vec{v} \sim \vec{\mathbf{eval}}\,\vec{u}\,\vec{v}}$$

Proof
By simultaneous induction over $t\ :\ \mathsf{Tm}\,\Gamma\,\sigma$ and $\vec{t}\ :\ \mathsf{Subst}\,\Gamma\,\Delta$. □

To show that quote maps equivalent values to equal normal forms, we have to simultaneously establish a dual property, as before for strong computability.

Lemma 12

$$\frac{v \sim_{\Gamma,\,\sigma} w}{\mathbf{quote}_{\Gamma,\,\sigma}\,v = \mathbf{quote}_{\Gamma,\,\sigma}\,w}(q) \qquad \frac{\overline{\mathbf{quote}}_{\Gamma,\,\sigma}\,m = \overline{\mathbf{quote}}_{\Gamma,\,\sigma}\,n}{m \sim_{\Gamma,\,\sigma} n}(u)$$

Proof
By induction over σ. For base types both properties follow directly from the definition of $\sim$ and the observation that all values of base type are neutral. We show both properties for $(\sigma{\rightarrow}\tau)$:

(q) Given $f \sim_{\Gamma,\,(\sigma\rightarrow\tau)} g$ (1) we have to show $\mathbf{quote}_{\Gamma,\,(\sigma\rightarrow\tau)}\,f = \mathbf{quote}_{\Gamma,\,(\sigma\rightarrow\tau)}\,g$. This reduces to showing $\lambda_\sigma \mathbf{quote}_{(\Gamma;\,\sigma),\,\tau}\,(f^{+}_{\sigma}@\varnothing) = \lambda_\sigma \mathbf{quote}_{(\Gamma;\,\sigma),\,\tau}\,(g^{+}_{\sigma}@\varnothing)$. Applying lemma 9 to (1) we obtain $f^{+}_{\sigma}{\sim}_{(\Gamma;\,\sigma),\,(\sigma\rightarrow\tau)}g^{+}_{\sigma}$ (2). Using ind.hyp. (u) for σ we can show $\varnothing{\sim}_{(\Gamma;\,\sigma),\sigma}\varnothing$ (3) and hence by the definition of $\sim$ and (2,3) we get $f^{+}_{\sigma}@\varnothing \sim_{(\Gamma;\,\sigma),\,\tau} f^{+}_{\sigma}@\varnothing$ (4). By applying ind.hyp (q) for τ to (4) we arrive at $\mathbf{quote}_{(\Gamma;\,\sigma),\,\tau}\,(f^{+}_{\sigma}@\varnothing) \sim_{(\Gamma;\,\sigma),\,\tau} \mathbf{quote}_{(\Gamma;\,\sigma),\,\tau}\,(f^{+}_{\sigma}@\varnothing)$.

(u) Given $\overline{\mathbf{quote}}_{\Gamma,\,(\sigma\rightarrow\tau)}\,m = \overline{\mathbf{quote}}_{\Gamma,\,(\sigma\rightarrow\tau)}\,n$ (1) we have to show $m \sim_{\Gamma,\,(\sigma\rightarrow\tau)} n$. Unfolding the definition of $\sim$ this means that given $v \sim_{(\Gamma\mathbin{+\!\!+}\Delta),\,\sigma} w$ (2) we have to show that $m^{+}_{\Delta}@v \sim_{(\Gamma\mathbin{+\!\!+}\Delta),\,\tau} n^{+}_{\Delta}@w$. Using the induction hypothesis (u) for τ this can be reduced to showing that $\overline{\mathbf{quote}}_{(\Gamma\mathbin{+\!\!+}\Delta),\,\tau}\,(m^{+}_{\Delta}\ v) = \overline{\mathbf{quote}}_{(\Gamma\mathbin{+\!\!+}\Delta),\,\tau}\,(n^{+}_{\Delta}\ w)$. This follows from (1) and $\mathbf{quote}_{\Gamma\mathbin{+\!\!+}\Delta,\sigma}\,v = \mathbf{quote}_{\Gamma\mathbin{+\!\!+}\Delta,\sigma}\,w$ which we can show by using ind.hyp (q) for σ with (2).

□

And also, we can exploit the second property to show that the identity environment is related to itself.

Corollary 13

$$\frac{x\ :\ \mathsf{Var}\,\Gamma\,\sigma}{x \sim x} \qquad \frac{\Gamma\ :\ \mathsf{Con}}{\mathbf{id}_\Gamma \sim \mathbf{id}_\Gamma}$$

We show the fundamental theorem of logical relations:

Theorem 14

$$\frac{t \simeq u \qquad \vec{v} \sim \vec{w}}{\mathbf{eval}\,t\,\vec{v} \sim \mathbf{eval}\,u\,\vec{w}} \qquad \frac{\vec{t} \simeq \vec{u} \qquad \vec{v} \sim \vec{w}}{\overrightarrow{\mathbf{eval}}\,\vec{t}\,\vec{v} \sim \overrightarrow{\mathbf{eval}}\,\vec{u}\,\vec{v}}$$

Proof
By mutual induction over the derivation of $t \simeq u$ and $\vec{t} \simeq \vec{u}$, as before we consider some typical cases. We assume that $\vec{v} \sim \vec{w}$(H).

refl, trans and sym Reflexivity follows from lemma 11, symmetry and transitivity from lemma 10.

ξ**:** To show $\mathbf{eval}\,(\lambda_\sigma\,t)\vec{v} \sim \mathbf{eval}\,(\lambda_\sigma\,u)\vec{w}$ it is sufficient to show $\lambda_\sigma\,t[\vec{v}] \sim_{\Gamma,\,(\sigma\to\tau)} \lambda_\sigma\,u[\vec{w}]$. Given $v \sim_{\Gamma +\!\!+ \Delta,\sigma} w$ we have to show that $\lambda_\sigma\,t[\vec{v}^{+}_{\Delta}]@v \sim_{\Gamma +\!\!+ \Delta,\tau} \lambda_\sigma\,u[\vec{w}^{+}_{\Delta}]@w$ which reduces to $\mathbf{eval}\,t\,(\vec{v}^{+}_{\Delta};\,v) \sim_{(\Gamma +\!\!+ \Delta),\,\sigma} \mathbf{eval}\,u\,(\vec{v}^{+}_{\Delta};\,w)$ this follows from the induction hypothesis, and lemma 9 applied to (H).

$\beta\sigma$**:** We have to show $\mathbf{eval}\,(((\lambda_\sigma\,t)[\vec{u}])\,u)\,\vec{v} \sim \mathbf{eval}\,(t[\vec{u};\,u])\,\vec{w}$. This reduces to having to show $\mathbf{eval}\,t\,(\overrightarrow{\mathbf{eval}}\,\vec{u}\,\vec{v};\mathbf{eval}\,u\,\vec{v}) \sim \mathbf{eval}\,t\,(\overrightarrow{\mathbf{eval}}\,\vec{u}\,\vec{w};\mathbf{eval}\,u\,\vec{w})$. This follows from applying lemma 11 to u and (H) to give (1), lemma 11 to $\vec{u}$ and (H) to give (2) and lemma 11 to t and (2,1).

assoc**:** We have to show $\overrightarrow{\mathbf{eval}}\,((\vec{s}\circ\vec{t})\circ\vec{u})\,\vec{v} \sim \overrightarrow{\mathbf{eval}}\,(\vec{s}\circ(\vec{t}\circ\vec{u}))\,\vec{w}$. This reduces to showing $\overrightarrow{\mathbf{eval}}\,\vec{s}\,(\overrightarrow{\mathbf{eval}}\,\vec{t}\,(\overrightarrow{\mathbf{eval}}\,\vec{u}\,\vec{v})) \sim \overrightarrow{\mathbf{eval}}\,\vec{s}\,(\overrightarrow{\mathbf{eval}}\,\vec{t}\,(\overrightarrow{\mathbf{eval}}\,\vec{u}\,\vec{w}))$ and this follows again by lemma 11: Applied first to $\vec{u}$ and (H) to give (1) then to $\vec{t}$ and (1) to give (2) and finally to $\vec{s}$ and (2).

□

By putting everything together we can establish soundness of the normalisation function:

Proposition 15

$$\frac{t \simeq u}{\mathbf{nf}\,t = \mathbf{nf}\,u}$$

Proof
Using corollary 13 and theorem 14 we can infer that $\mathbf{eval}\,t\,\mathbf{id} \sim \mathbf{eval}\,u\,\mathbf{id}$ and hence by lemma 12 we obtain the result. □

7 System T

It is straightforward to extent our system to include a type of natural numbers. We replace the base type $\bullet$ with N and extend the syntax of terms with zero $\mathsf{0}$, successor suc and primitive recursion prec.

$$\frac{}{\mathsf{0} \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N}} \qquad \frac{t \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N}}{\mathsf{suc}\,t \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N}} \qquad \frac{n \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N} \quad f \;:\; \mathsf{Tm}\,\Gamma\,\mathsf{N}{\to}\sigma{\to}\sigma \quad z \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathsf{prec}\,n\,f\,z \;:\; \mathsf{Tm}\,\Gamma\,\sigma}$$

We add the following $\simeq$ rules to the equational theory (and congruences for suc and prec):

$$\begin{array}{lcll} \mathsf{prec}\,\mathsf{0}\,f\,z & \simeq & z & \mathsf{cprimrecz} \\ \mathsf{prec}\,(\mathsf{suc}\,n)\,f\,vz & \simeq & f\,n\,(\mathsf{prec}\,n\,f\,z) & \mathsf{cprimrecs} \end{array}$$

Values Val and normal forms are extended with $\mathsf{0}$ and suc and neutral terms Ne with a constructor to represent primitive recursion applied to a neutral natural number:

$$\frac{}{\mathsf{0} \;:\; \mathsf{Val}\,\Gamma\,\mathsf{N}} \qquad \frac{v \;:\; \mathsf{Val}\,\Gamma\,\mathsf{N}}{\mathsf{suc}\,v \;:\; \mathsf{Val}\,\Gamma\,\mathsf{N}} \qquad \frac{}{\mathsf{0} \;:\; \mathsf{Nf}\,\Gamma\,\mathsf{N}} \qquad \frac{n \;:\; \mathsf{Nf}\,\Gamma\,\mathsf{N}}{\mathsf{suc}\,n \;:\; \mathsf{Nf}\,\Gamma\,\mathsf{N}}$$

$$\frac{n \;:\; \mathsf{Ne}^{T}\,\Gamma\,\mathsf{N} \quad f \;:\; \mathsf{Val}\,\Gamma\,(\mathsf{N}{\to}\sigma{\to}\sigma) \quad z \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathsf{prec}\,n\,f\,z \;:\; \mathsf{Ne}^{T}\,\Gamma\,\sigma}$$

A separate semantic primitive recursor **pr** is added and **eval** extended to accommodate it.

$$\frac{n \;:\; \mathsf{Val}\,\Gamma\,\mathsf{N} \quad f \;:\; \mathsf{Val}\,\Gamma\,(\mathsf{N}{\to}\sigma{\to}\sigma) \quad z \;:\; \mathsf{Val}\,\Gamma\,\sigma}{\mathbf{pr}\,n\,f\,z \;:\; \mathsf{Val}\,\Gamma\,\sigma}$$

$$\begin{array}{lllcll} \mathbf{pr} & \mathsf{0} & f & z & \Rightarrow & z \\ \mathbf{pr} & (\mathsf{suc}\,n) & f & z & \Rightarrow & f\,@\,n\,@\,(\mathbf{pr}\,n\,f\,z) \end{array}$$

$$\begin{array}{llcll} \mathbf{eval} & \mathsf{0} & \vec{v} & \Rightarrow & \mathsf{0} \\ \mathbf{eval} & (\mathsf{suc}\,n) & \vec{v} & \Rightarrow & \mathsf{suc}\,(\mathbf{eval}\,n\,\vec{v}) \\ \mathbf{eval} & (\mathsf{prec}\,n\,f\,z) & \vec{v} & \Rightarrow & \mathbf{pr}\,(\mathbf{eval}\,n\,\vec{v})\,(\mathbf{eval}\,f\,n)\,(\mathbf{eval}\,z\,n) \end{array}$$

For **quote** we replace the case for $\mathbf{quote}_{\bullet}$ with cases for $\mathbf{quote}_{\mathsf{N}}$

$$\begin{array}{llcl} \mathbf{quote}_{\mathsf{N}} & \mathsf{0} & \Rightarrow & \mathsf{0} \\ \mathbf{quote}_{\mathsf{N}} & (\mathsf{suc}\,n) & \Rightarrow & \mathsf{suc}\,(\mathbf{quote}_{\mathsf{N}}\,n) \\ \mathbf{quote}_{\mathsf{N}} & n & \Rightarrow & \overline{\mathbf{quote}}\,n \end{array}$$

and the big-step semantics is updated accordingly. Next we replace the base cases $\bullet$ in the definitions of SCV and $\sim$ with inductively defined notions for N.

$$\frac{}{\mathsf{SCV}_{\Gamma,\mathsf{N}}\,\mathsf{0}} \qquad \frac{\mathsf{SCV}_{\Gamma,\mathsf{N}}\,n}{\mathsf{SCV}_{\Gamma,\mathsf{N}}\,(\mathsf{suc}\,n)} \qquad \frac{\overline{\mathsf{quote}}\,n \Downarrow m \quad \ulcorner n \urcorner \simeq \ulcorner m \urcorner}{\mathsf{SCV}_{\Gamma,\mathsf{N}}\,n}$$

$$\frac{}{\mathsf{0} \sim_{\mathsf{N}} \mathsf{0}} \qquad \frac{m \sim_{\mathsf{N}} n}{(\mathsf{suc}\,m) \sim_{\mathsf{N}} (\mathsf{suc}\,n)} \qquad \frac{\overline{\mathbf{quote}}\,m = \overline{\mathbf{quote}}\,n}{m \sim_{\Gamma,\mathsf{N}} n}$$

We also require an extra lemma to prove the fundamental theorem:

Lemma 16

$$\frac{\mathsf{SCV}_{\Gamma,\,(\mathsf{N}\to\sigma\to\sigma)}\,f \quad \mathsf{SCV}_{\Gamma\,\sigma}\,z \quad \mathsf{SCV}_{\Gamma,\,\mathsf{N}}\,n}{\Sigma v : \mathsf{Val}\,\Gamma\,\sigma\,.\,\mathsf{pr}\,f\;z\;n \Downarrow v \wedge \mathsf{prec}\,\ulcorner f\urcorner\ulcorner z\urcorner\ulcorner n\urcorner \simeq \ulcorner v\urcorner \wedge \mathsf{SCV}\,v}$$

Proof
By induction over $\mathsf{SCV}_{\Gamma,\,\mathsf{N}}\,n$. □

8 Conclusions

Let us summarize the main result of this paper:

Theorem 17
We have defined a function in total Type Theory:

$$\frac{t \;:\; \mathsf{Tm}\,\Gamma\,\sigma}{\mathbf{nf}\,t \;:\; \mathsf{Nf}\,\Gamma\,\sigma}$$

with the following properties:

$$\begin{array}{ll} \textbf{soundness} & \dfrac{t \;\simeq\; t'}{\mathbf{nf}\,t \;=\mathbf{nf}\,t'} \\ \textbf{completeness} & \dfrac{}{t \;\simeq\; \mathbf{nf}\,t} \\ \textbf{stability} & \dfrac{n \;:\; \mathsf{Nf}\,\Gamma\,\sigma}{\mathbf{nf}\,n \;=\; n} \end{array}$$

Proof
Propositions 7 and 15. □

As we have already indicated, we choose the names, because we consider normal forms as a syntactic model construction. Moreover, the 2nd property, completeness, implies that the inverse of soundness holds:

Corollary 18

$$\frac{\mathbf{nf}\,t = \mathbf{nf}\,u}{t \simeq u}$$

Since our definition of normal form is a first order inductive definition (see p. 3), it is clear that equality of normal forms is decidable. Hence, we obtain the following corollary:

Corollary 19
Given $t, u \;:\; \mathsf{Tm}\,\Gamma\,\sigma$, it is decidable whether $t \simeq u$ holds.

Moreover, the last property, stability, clearly implies that **nf** is surjective on normal forms. As a consequence, we can prove relevant properties of terms by induction over normal forms.

This is not a new result: it can be obtained by proving Strong Normalisation of a suitably chosen small-step reduction relation (avoiding Mellies' problem) or by using normalisation by evaluation. What have we gained by our approach?

First of all, the traditional approach using term-rewriting does not directly lead to a realistic implementation of normalisation. We can use strong normalisation to justify such an implementation but this requires yet another proof. Also we wonder why we have to first fight with the non-determinism introduced by the small-step relation only to throw it away in the end anyway. The case of the typed λ-calculus with strong sums (Lindley, 2007) is a good example. Lindley's excellent analysis clearly suggests an algorithm to compute normal forms, but this is lost due to the need to fit it in the framework of term-rewriting.

Second, the term-rewriting approach means that we need to prove the Church-Rosser property as a property of our rewriting system. In our experience, it often requires a fair amount of ingenuity to have Church-Rosser without loosing completeness or strong normalisation. In our setting, equational soundness is shown by using the fundamental property of logical relation. This at least inspires some hope that our construction will be more easily generalizable to other calculi.

What about Normalisation by Evaluation (NbE)? In our view, see (Altenkirch *et al.*, 1995), NbE is basically a semantic construction: we provide a complete model construction, we show completeness by constructively inverting evaluation. NbE gives us a beautiful high-level analysis of normalisation, however, it's actual computational content is often not immediately clear, the normalisation functions seems to work by magic. No doubt this counterintuitive nature of NbE has a lot to do with the intensive use of higher order functions in it's implementation. They are also the reason that NbE can be only formalized in a metatheory where constructive higher-order functions are a primitive. This may be one reason why the traditional approach using term-rewriting is still more popular: it can be easily formulated within standard set theory.

It has been suggested by one anonymous referee that we should try to derive our algorithm from NbE together with the implementation of an evaluator for the functional functional metalanguage which is used to execute NbE. It seems plausible that this is possible, since this is the obvious way to eliminate the higher order character of NbE. However, we doubt that much is gained by doing so, because we claim that our approach has its own intuitive beauty and doesn't need to be justified by translation. Let's look back at what we have done!

How does one implement normalisation? We reduce to values, corresponding to weak normal forms and iterate the process (**quote**). This gives rise to a normalizer to β-normal forms. The only modification required for η in this case is to recursively η-expand every functional term. How do we prove termination: basically by adopting Tait's method of strengthening the induction hypothesis to function types — i.e. by using logical predicates. Since we go under λs we really need Kripke logical predicates — this is traditionally swept under the carpet by syntactic trickery using an infinite supply of fresh variables. Completeness, i.e. the result of normalisation is convertible to it's input can be shown at the same time since it follows the same logical structure. How do we prove soundness? We use logical relations, indeed Kripke logical relations for the same reasons as above. In the present paper we have spelled out the details of this construction in great detail corresponding to a formalisation using the Agda system (Chapman, 2007) — some of its elegance may have got lost in the details.

Our recipe, we believe, is applicable to many cases, avoiding syntactic hackery on the one hand and high-level magic on the other. Clearly, we have to justify this claim by actually applying our method to well known difficult cases: typed λ-calculus with sums or other extensions such as bar-recursion, dependent types with η rules, and the combination, i.e. dependent types with non-empty sums and bar-recursion, the latter are calculi whose metatheoretic properties are not yet established.

References

Abadi, Martín, Cardelli, Luca, Curien, Pierre-Louis, & Lèvy, Jean-Jacques. (1990). Explicit Substitutions. *Pages 31–46 of: Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California.* ACM.

Altenkirch, Thorsten, & Chapman, James. (2006). Tait in one big step. *Workshop on Mathematically Structured Functional Programming, MSFP 2006, Kuressaare, Estonia, July 2, 2006.* electronic Workshop in Computing (eWiC). Kuressaare, Estonia: The British Computer Society (BCS).

Altenkirch, Thorsten, Hofmann, Martin, & Streicher, Thomas. (1995). Categorical reconstruction of a reduction free normalization proof. *Pages 182–199 of:* Pitt, David, Rydeheard, David E., & Johnstone, Peter (eds), *Category theory and computer science.* LNCS 953.

Altenkirch, Thorsten, Dybjer, Peter, Hofmann, Martin, & Scott, Phil. (2001). Normalization by evaluation for typed lambda calculus with coproducts. *Pages 303–310 of: 16th annual ieee symposium on logic in computer science.*

Balat, Vincent. (2002). *Une étude des sommes fortes : isomorphismes et formes normales.* Ph.D. thesis, Université Denis Diderot.

Berger, Ulrich, & Schwichtenberg, Helmut. (1991). An inverse of the evaluation functional for typed λ–calculus. *Pages 203–211 of:* Vemuri, R. (ed), *Proceedings of the sixth annual ieee symposium on logic in computer science.* IEEE Computer Science Press, Los Alamitos.

Bove, Ana, & Capretta, Venanzio. (2001). Nested general recursion and partiality in type theory. *Pages 121–135 of:* Boulton, Richard J., & Jackson, Paul B. (eds), *Theorem proving in higher order logics: 14th international conference, tphols 2001.* Lecture Notes in Computer Science, vol. 2152. Springer-Verlag.

Chapman, James. (2007). *Formalisation of BSN for System T.* `http://www.cs.nott.ac.uk/~jmc/BSN.html`.

Coquand, Catarina. (2002). A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions. *Higher order symbol. comput.*, **15**(1), 57–90.

Coquand, Thierry. (1991). An algorithm for testing conversion in type theory. New York, NY, USA: Cambridge University Press.

David, Rene. (2001). Normalization without reducibility. *Annals of pure and applied logic*, **107**(1-3), 121–130.

Girard, J.-Y., Lafont, Y., & Taylor, P. (1989). *Proofs and Types.* Cambridge University Press.

Jay, C. Barry, & Ghani, Neil. (1995). The virtues of eta-expansion. *Journal of functional programming*, **5**(2), 135–154.

Levy, Paul Blain. (2001). *Call-by-push-value.* Ph.D. thesis, Queen Mary, University of London.

Lindley, S. (2007). Extensional Rewriting with Sums. *Pages 255–271 of: Typed Lambda Calculus and Applications*. lncs, vol. 4583. Springer.

McBride, Conor. (2005a). *Epigram*. `http://www.e-pig.org`.

McBride, Conor. (2005b). Epigram: Practical programming with dependent types. *Pages 130–170 of:* Vene, Varmo, & Uustalu, Tarmo (eds), *Advanced Functional Programming 2004*. Lecture Notes in Computer Science, vol. 3622. Springer-Verlag. Revised lecture notes from the International Summer School in Tartu, Estonia.

McBride, Conor, & McKinna, James. (2004). The view from the left. *Journal of functional programming*, **14**(1).

Melliès, Paul-André. (1995). Typed lambda-calculi with explicit substitutions may not terminate. *Pages 328–334 of: Tlca '95: Proceedings of the second international conference on typed lambda calculi and applications*. London, UK: Springer-Verlag.

Norell, Ulf. (2007a). *Agda 2*. `http://www.cs.chalmers.se/~ulfn/`.

Norell, Ulf. 2007b (September). *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Tait, William W. (1967). Intensional interpretations of functionals of finite type. *Journal of symbolic logic*, **32**, 198–212.