λ-calculus and types

Lecture notes
Midland Graduate School / APPSEM Spring School 2004

Thorsten Altenkirch

School of Computer Science and Information Technology, Nottingham University
`txa@cs.nott.ac.uk`

1 Introduction

1.1 A short history

The λ-calculus was introduced by Alonzo Church in the 40ies. The initial goal was formalisation of mathematical reasoning, I believe. Then Turing showed that the λ-definable functions on encodings of natural numbers are the same as the ones definable by Turing-machines. This lead to the *Church-Turing thesis* that all notions of computations are equivalent in power.

Realizing the potential of λ-calculus for programming, in the 50ies, McCarthy used λ-notation in the design of the programming language LISP (for LISt Processing). This started the tradition of *functional programming languages*, which lead to languages such as ML in the 80ies and Haskell in the 90ies. Both language are based on typed versions of the λ-calculus. ML following the LISP tradition is an impure functional languages, allowing effects such that updates of the memory to occur during evaluation and further computation depend on this, while Haskell is pure: a function is applied to the same value will always return the same answer.

With hindsight we may say that one of the main stumbling blocks of the early work on λ-calculus was the concentration on the untyped λ-calculus, which made it hard to use to codify reasoning due to the existence of *fixpoint combinators*, which allow the construction of the fixpoint of any function, e.g. negation, which leads to logical paradoxes. Indeed, it was only in the 70ies when Dana Scott found a semantic explanation of the untyped λ-calculus, introducing *domain theory* which can be used as a tool to model and reason about computations semantically.

Typed λ-calculus was introduced by Curry and Church, this system has a much simpler semantic interpretation in terms of total functions and it can be applied in programming to avoid the occurence of run-time errors. It was realized by Curry and Howard that there is an interesting link between λ-calculus and logic: typed λ-terms can be viewed as proofs in intuitionistic logic — this is called the *Curry-Howard-isomorphism.* Another interesting connection was discovered first by Lambek: typed λ-calculus can be viewed as the theory of cartesian closed categories (CCCs).

In the eary 70ies Girard introduced *System F*, a typed λ-calculus to model 2nd order logic. His strong normalisation proof can be viewed as a proof of Takeuti's conjecture, that there is a purely syntactical consistency proof of 2nd order logic. It is noticable that basically the same calculus was rediscovered by Reynolds to model polymorphic functions in computer science. System F had a big impact on the analysis and design of modern programming languages, e.g. see Benjamin Pierce's recent book [9] for a comprehensive overview of types for programming languages. However, System F comes with some semantic troubles, e.g. Reynolds had to realize that there is no simple set-theoretic model of System F and the strong normalisation proof requires impredicative reasoning, which makes it less interesting for foundational purposes.

Also in the early 70ies Per Martin-Löf started the development of Type Theory, a constructive foundation of mathematics, bringing the Curry-Howard-isomorphism to its logical conclusion by identifying types and propostions (and consequently programs and proofs). A central concept is the notion of a dependent type, i.e. a type which can depend on values. Type Theory and its impredicative extension the Calculus of Constructions by Coquand and Huet (basically a combination of System F and Type Theory) lead to a number of computer aided reasoning tools such as NuPRL, LEGO, COQ and ALF. We are now beginning to realize the potential of Type Theory for the design of programming languages and program development tools for the proliferation of programs, which are correct by construction. This topic will be explored in Conor McBride's course on *Dependently Typed Programming* using his recent language and tool *Epigram.*

I apologize for the lack of references in this section. I hope to be able to fix this in future versions of these notes.

1.2 Ideology and programme

These notes are based the view that typed λ-calculi should be considered as fundamental because they can be used to model total functions which are easier to understand than partial functions and which are ultimatly more useful to construct solutions of problems.

What is a (total) function? In Set Theory functions are represented as a relation which is total on the domain and deterministic on the codomain. In my view this does not reflect my intuitive understanding of a function as a mechanism, to which I can input values of the domain and which will reply with an element of the codomain. An important aspect of the constructive idea of a function is that while a function is given by a mechanism, we are not allowed to inspect it, i.e. the mechanism is hidden within a black box. This theory of a functions can be formalized by the $\beta\eta$-equality of typed λ calculi. If we are only interested in computation but not in the consequences of the black box view, the weaker β-equality is sufficient.

Given my views on traditional Set Theory, it may not be surprising that the development presented in these notes takes place in a constructive metatheory. However, I will use familiar notations and notions, whenever possible. Having said this I am quite fond of a two-dimensional syntax, which uses rules to represent constructions in the conclusion relying on the presence of a number of objects given in the premise.

I write $A\,\mathbf{Set}$ for A is a set and $P\,\mathbf{Prop}$ for P is a proposition and I write $a \in A$ for a is an element of A. A central means of defining sets and propositions are inductive definitions which show how elements of sets can be constructed and what propositions can be proven.

1.3 Overview

In section 2 I will introduce the simply typed λ calculus and will discuss some core issues, such as substitution and α-equality, β and $\beta\eta$-equality and the relation to cartesian closed categories, combinatory logic, functional encodings and primitive datatypes like Bool or Nat. In section 3, I'll get a bit more technical and present two proofs of normalisation for β-equality: *normalisation by reduction* and *normalisation by evaluation.* I plan to add a section on *normalisation by evaluation* for $\beta\eta$-equality in a future version of these notes.

2 The simply typed λ-calculus

In the following we will introduce the simply typed λ-calculus, $\lambda^{\rightarrow}_{\mathcal{X}}$ over a given set of type variables or uninterpreted base types $\mathcal{X}$. In the literatur frequently the calculus with just one base type $\circ$ is considered, we would write this as $\lambda^{\rightarrow}_{\{\circ\}}$. To avoid being restricted to functional encodings of data we will also consider the simply typed λ calculus with a type of Booleans $\lambda^{\rightarrow 2}$ or with a type of natural numbers $\lambda^{\rightarrow N}$, the latter is commonly called *System T* and is closely related to arithmetic. Since all our definitions are modular we may consider the calculus with Booleans, natural numbers and type variables $\lambda^{\rightarrow 2N}_{\mathcal{X}}$. However, some theorems, e.g. the decidability of $\beta\eta$-equality does only hold for some of the calculi (and the proofs are be different for different calculi).

2.1 Types and terms

All the following definitions are indexed by a finite finite set of type variables $\mathcal{X}$, we use X, Y, Z for variables ranging over $\mathcal{X}$. Typical members of $\mathcal{X}$ include $\mathtt{X}, \mathtt{Y}, tZ$. Given this we define the set of pure types $\mathrm{Ty}_{\mathcal{X}}$.

$$\frac{X \in \mathcal{X}}{X \in \mathrm{Ty}_{\mathcal{X}}} \qquad \frac{\sigma, \tau \in \mathrm{Ty}_{\mathcal{X}}}{\sigma \to \tau \in \mathrm{Ty}_{\mathcal{X}}}$$

We are going to omit the index $\mathcal{X}$ from now on, and indeed we will in general omit indizes which can be inferred from the context. We are overloading the meta-level notation for function space $\to$ and the symbol for function types, and again we will overload more notations (such as the empty space representing application) to avoid the proliferation of different symbols and names. When writing types, we follow the convention that $\to$ is right-associative, i.e. $\sigma \to \tau \to \rho$ reads as $\sigma \to (\tau \to \rho)$.

To introduce terms we assume as given an infinite set of term variables $\mathcal{V}$ with a decidable equality, i.e. we assume that given $x, y \in \mathcal{V}$ we can constructively show $x = y \vee x \neq y$. We use x, y, z for variables ranging over $\mathcal{V}$ and $\mathtt{x}, \mathtt{y}, \mathtt{z}$ are typical members of $\mathcal{V}$. In an implementation we are using `String` to represent $\mathcal{V}$. Infinity is witnessed by a function fresh which to any finite set of variables $P \subseteq_{fin} \mathcal{V}$ assigns $\mathrm{fresh}\, P \in \mathcal{V} \setminus P$, that is $\mathrm{fresh} P \notin P$.

We introduce contexts Con which assign types to a finite set of free variables, Con is defined inductively by

$$\frac{}{() \in \mathrm{Con}} \qquad \frac{\Gamma \in \mathrm{Con}, x \in \mathcal{V}, \sigma \in \mathrm{Ty}}{\Gamma x^\sigma \in \mathrm{Con}}$$

We write contexts as $x_1^{\sigma_1} x_2^{\sigma_2} \dots x_n^{\sigma_n}$ omitting the initial empty context $()$. We define define the set of variables and terms of type σ in context Γ, $\mathrm{Var}_\Gamma\, \sigma \subseteq \mathrm{Tm}_\Gamma\, \sigma$, inductively, starting with the rules for variables:

$$\frac{}{x \in \mathrm{Var}_{\Gamma x^\sigma}\, \sigma} \qquad \frac{x \in \mathrm{Var}_\Gamma\, \sigma \quad x \neq y}{x \in \mathrm{Var}_{\Gamma . y^\tau}\, \sigma}$$

The side condition in the second rule is essential to handle shadowing, i.e. $x \notin \mathrm{Tm}_{x^\sigma x^\tau}\, \sigma$ if $\sigma \neq \tau$. Basically we view contexts as functions $P \to \mathrm{Ty}$ for a finite $P \subseteq_{fin} \mathcal{V}$.

$$\frac{t \in \mathrm{Tm}_\Gamma\, \sigma \to \tau \quad u \in \mathrm{Tm}_\Gamma\, \sigma}{t\, u \in \mathrm{Tm}_\Gamma\, \sigma} \qquad \frac{t \in \mathrm{Tm}_{\Gamma x : \sigma}\, \tau}{\lambda x^\sigma . t \in \mathrm{Tm}_\Gamma\, \sigma \to \tau}$$

When writing terms we adopt the following conventions: application is left-associative, e.g. $t\, u\, v$ reads as $(t\, u)\, v$; and the scope of λ extends as far as possible, e.g. $\lambda x^\sigma . t\, u$ reads as $\lambda x^\sigma . (t\, u)$. Beware, there are a lot of different conventions and styles used in the literature.

We will omit type or context if it can be easily inferred form the context.

In the literature $t \in \mathrm{Tm}_\Gamma\, \sigma$ is usually written as $\Gamma \vdash t : \sigma$ - however this suggests that there is a notion of untyped terms which are sorted. Our view is that typed terms are fundamental, hence the definition as a family, i.e. a set indexed by contexts and types.

2.2 α-equality and substitution

We have introduced a named syntax as a way to refer to a binding, this syntax is too intensional, i.e. $\lambda \mathtt{x}^{\sigma \to \tau} . \lambda \mathtt{y}^\sigma . \mathtt{xy}$ is different from $\lambda \mathtt{a}^{\sigma \to \tau} . \lambda \mathtt{b}^\sigma . \mathtt{ab}$, even though they have the same binding structure. To remedy this, we introduce the notion of α-equality $=_\alpha$ on typed terms, e.g. $\lambda \mathtt{x}^{\sigma \to \tau} . \lambda \mathtt{y}^\sigma . \mathtt{xy} =_\alpha \lambda \mathtt{a}^{\sigma \to \tau} . \lambda \mathtt{b}^\sigma . \mathtt{ab}$. We define α-equality for terms of the same type but in different contexts wrt to a set of identifications of variables P inductively:

$$\frac{(x, y) \in P}{x =_\alpha^P y} \qquad \frac{t =_\alpha^P u \quad t' =_\alpha^P u'}{t\, u =_\alpha^P t'\, u'} \qquad \frac{t =_\alpha^P \cup \{(x, y)\} u}{\lambda x^\sigma . t =_\alpha^P \lambda y^\sigma . u}$$

While it is convenient to define $=_\alpha$ wrt P in the end we are only interested in $P = \{(x,x) \mid x \in \mathrm{Var}_\Gamma\,\sigma\}$ and on terms in the same contexts, which we denote by $=_\alpha$. While the basic idea of α-equivalence seems obvious, the details can be daunting. See my unpublished pearl [3] which discusses α-equivalence in an untyped setting. There I also show that $=_\alpha$ is an equivalence relation (reflexive, symmetric and transitive) – this proof can be easily transferred to a typed setting.

An alternative to named terms are de-Bruijn-terms, where variables are replaced by numbers, indicating binding depth. Consequently λ just introduces an unnamed abstraction. Both $\lambda\mathtt{x}^{\sigma\to\tau}.\lambda\mathtt{y}^{\sigma}.\mathtt{xy}$ and $\lambda\mathtt{a}^{\sigma\to\tau}.\lambda\mathtt{b}^{\sigma}.\mathtt{ab}$ become $\lambda^{\sigma\to\tau}\lambda^{\sigma}1\,0$. In general deBrujn terms are representations of α-equivalence classes.

To introduce β-equality $(\lambda x^\sigma.t)\,u =_\beta t[x=u]$ we need substitution. While it is standard to introduce only substitution of one variable, I find it is technically better to take parallel substitution as primitive. For that purpose I define $\mathrm{Subst}_\Delta\,\Gamma$ which assigns to all variables in Γ terms typable in Δ of the appropriate types.

$$\frac{}{() \in \mathrm{Subst}_\Delta\,()} \qquad \frac{\vec{t} \in \mathrm{Subst}_\Delta\,\Gamma \quad t \in \mathrm{Tm}_\Delta\,\sigma}{(\vec{t}, x = t) \in \mathrm{Subst}_\Delta\,\Gamma.x:\sigma}$$

Given $\vec{t} \in \mathrm{Subst}_\Delta\,\Gamma$ and $x \in \mathrm{Var}_\Gamma\,\sigma$ we define $x[\vec{t}] \in \mathrm{Tm}_\Delta\,\sigma$

$$x[\vec{t}, y = t] = \begin{cases} t & \text{, if } x = y \\ x[\vec{t}] & \text{, if, if } x \neq y \end{cases}$$

Why is the case $x[()]$ missing? Simply, because there aren't any free variables typable in the empty context. We continue by extending substitution to terms $t \in \mathrm{Tm}_\Gamma\,\sigma$:

$$\begin{aligned} (t\,u)[\vec{t}] &= t[\vec{t}]\,u[\vec{t}] \\ (\lambda x^\sigma.t)[\vec{t}] &= \lambda y^\sigma.t[\vec{t}, x = y] \quad \text{where } y = \mathrm{fresh}\,\Gamma \end{aligned}$$

By $\mathrm{fresh}\,\Gamma$ we mean the application of the previously introduced function fresh to the set of variables mentioned in Γ. Using a fresh variable here is essential to achieve that substitution preserves α-equality. E.g. if we would naively substitute, this may fail: we know that $\lambda\mathtt{x}^\sigma.\mathtt{y} =_\alpha \lambda\mathtt{z}^\sigma.\mathtt{y}$, now if $(\lambda\mathtt{z}^\sigma.\mathtt{y})[\mathtt{y} = \mathtt{z}]$ were $\lambda\mathtt{z}^\sigma.\mathtt{z}$ we had $(\lambda\mathtt{x}^\sigma.\mathtt{y}[\mathtt{y} = \mathtt{z}]) =_\alpha \lambda\mathtt{x}^\sigma.\mathtt{z} \neq_\alpha \lambda\mathtt{z}^\sigma.\mathtt{z}$. However, according to our definition $(\lambda\mathtt{z}^\sigma.\mathtt{y})[\mathtt{y} = \mathtt{z}] = \lambda z'.\mathtt{z}$ where z' is the fresh variable chosen. Obviously, we have to verify in detail that $=_\alpha$ is always preserved, here I refer again to [3].

Subsequently, we will consider only operations on terms upto $=_\alpha$, that is we are working with the quotient $\mathrm{Tm}^\alpha_\Gamma\,\sigma = (\mathrm{Tm}_\Gamma\,\sigma)/=_\alpha$. To avoid the proliferation of αs everywhere we just write $\mathrm{Tm}_\Gamma\,\sigma$ and never do anything which is not closed under α-equivalence.

We can compose substitutions by iterating application:

$$\frac{\vec{t} \in \mathrm{Subst}_\Delta\,\Gamma \quad \vec{u} \in \mathrm{Tm}_\Gamma\,\Theta}{\vec{t} \circ \vec{u} \in \mathrm{Subst}_\Delta\,\Theta}$$

$$\begin{aligned} \vec{t} \circ () &= () \\ \vec{t} \circ (\vec{u}, x = u) &= (\vec{t} \circ \vec{u}, x = u[\vec{t}], \vec{t} \circ \vec{u}) \end{aligned}$$

Moreover we can define $1_\Gamma \in \mathrm{Subst}_\Gamma\, \Gamma$:

$$\begin{aligned} 1_{()} &= () \\ 1_{\Gamma.x:\sigma} &= (1_\Gamma, x = x) \end{aligned}$$

And it shouldn't come as a surprise that this is a category, e.g. that

$$\begin{aligned} 1 \circ \vec{u} &= \vec{u} \\ \vec{t} \circ 1 &= \vec{t} \\ (\vec{t} \circ \vec{v}) \circ \vec{v} &= \vec{t} \circ (\vec{u} \circ \vec{v}) \end{aligned}$$

I leave it as an exercise to fill in the missing sets (e.g. where do $\vec{t}, \vec{u}, \vec{v}$ live) and the indizes to 1.

We can now define the substitution of a single variable as a special case: given $u \in \mathrm{Tm}_\Gamma\, \sigma$ we have $(1, x = u) \in \mathrm{Subst}_\Gamma\, \Gamma x : \sigma$. Hence, given $t \in \mathrm{Tm}_{\Gamma x:\sigma}\, \tau$ we have $t[x = u] = t[(1, x = u)] \in \mathrm{Tm}_\Gamma\, \tau$.

Actually, I am cheating a little bit since the variable we want to substitute may not be at the end of the context. I am sure the reader can fix this little problem.

2.3 β- and $\beta\eta$-equality

We will introduce two equalities on typed terms: β-equality $=_\beta$ and $\beta\eta$-equality $=_{\beta\eta}$. While the weaker, intensional β-equality captures the idea that we can execute functions and evaluate their arguments, the stronger, extensional $\beta\eta$-equality also captures the idea that a function is a black box.

For any terms $t, u \in \mathrm{Tm}_\Gamma\, \sigma$ we introduce $t =_\beta^{\Gamma,\sigma} u$ inductively as

$$\frac{}{(\lambda x^\sigma.t)\, u =_\beta t[x = u]}(\beta) \qquad \frac{t =_\beta t' \quad u =_\beta u'}{t\, u =_\beta t'\, u'}(\mathrm{app})$$

To make sure that $=_\beta$ is at least an equivalence relation we also add

$$\frac{}{t =_\beta t}(\mathrm{refl}) \qquad \frac{t =_\beta u}{u =_\beta v}(\mathrm{sym}) \qquad \frac{t =_\beta u \quad u =_\beta v}{t =_\beta v}(\mathrm{trans})$$

Note that I am omitting types and contexts to improve readability, it shouldn't be hard to fill them in. $=_{\beta\eta}$ is defined by the same rules and additionally

$$\frac{t =_{\beta\eta}^{\Gamma.x:\sigma,\tau} u}{\lambda x^\sigma.t =_{\beta\eta} \lambda x^\sigma.u}(\xi) \qquad \frac{t \in \mathrm{Tm}_\Gamma\, \sigma \to \tau \quad z = \mathrm{fresh}\, \Gamma}{\lambda z^\sigma.t\, z =_{\beta\eta} t}(\eta)$$

In the literature $=_\beta$ usually refers to the $=_\beta + (\xi)$. In my opinion this is a mongrel, neither intensional nor extensional. Its central role in λ-calculus can only be justified historically. What I call $=_\beta$ is sometimes called weak β-equality.

If we consider terms and substitutions upto $\beta\eta$-equality the category of contexts and substitution becomes a cartesian closed category, short CCC. Indeed it is an initial CCC, that is only the equations necessary to make it cartesian closed hold.

Let me explain what this means, to do this without too much hacking I will bend the rules of category theory a little bit. A type σ corresponds to a context of length 1: x^σ for any variable x. Now given a context Γ and a type σ we can construct $\Gamma \times \sigma = \Gamma.x : \sigma$, where $x = \mathrm{fresh}\,\Gamma$. Indeed, this is a product, we can establish an isomorphism

$$\phi_\times \in \mathrm{Subst}_\Delta\,\Gamma \times \sigma \cong (\mathrm{Subst}_\Delta\,\Gamma) \times (\mathrm{Tm}_\Gamma\,\sigma)$$

by

$$\begin{aligned}
\phi_\times &\in \mathrm{Subst}_\Delta\,\Gamma \times \sigma \to (\mathrm{Subst}_\Delta\,\Gamma) \times (\mathrm{Tm}_\Gamma\,\sigma)\\
\phi_\times\,(\vec{t}, x = t) &= (\vec{t}, t)\\
\phi_\times^{-1} &\in (\mathrm{Subst}\ \Delta\,\Gamma) \times (\mathrm{Tm}\ \Gamma\,\sigma) \to (\mathrm{Subst}\ \Delta\,\Gamma) \times (\mathrm{Tm}\ \Gamma\,\sigma)\\
\phi_\times^{-1}\,(\vec{t}, t) &= (\vec{t}, x = t)
\end{aligned}$$

This looks as if nothing is happening (and it is easy to see that $\phi \circ \phi^{-1}$ and $\phi^{-1} \circ \phi$ are identities) However, notice that the $\times$ on the left hand side is our definition of products in the category of substitutions, while the $\times$ on the right hand side is the set-theoretic product.

So far we haven't used $=_{\beta\eta}$ but only the definition of substitution. However, cartesian closure means that there is a right adjoint to $\Gamma \times \sigma$, that is we can establish:

$$\phi_\to \in \mathrm{Tm}^{\beta\eta}_{\Gamma\times\sigma}\,\tau \cong \mathrm{Tm}^{\beta\eta}_{\Gamma}\,\sigma \to \tau$$

i.e. we have

$$\begin{aligned}
\phi_\to &\in \mathrm{Tm}^{\beta\eta}_{\Gamma\times\sigma}\,\tau \cong \mathrm{Tm}^{\beta\eta}_{\Gamma}\,\sigma \to \tau\\
\phi_\to\,t &= \lambda x^\sigma.t\\
\phi_\to^{-1} &\in \mathrm{Tm}^{\beta\eta}_{\Gamma}\,\sigma \to \tau \to \mathrm{Tm}^{\beta\eta}_{\Gamma\times\sigma}\,\tau\\
\phi_\to^{-1}\,u &= u\,x
\end{aligned}$$

We have to check that both compositions are identities:

$$\begin{aligned}
\phi^{-1}(\phi\,t) &= (\lambda x^\sigma.t)\,x\\
&=_{\beta\eta} t\\
\phi(\phi^{-1}\,u) &= \lambda x^\sigma.u\,x\\
&=_{\beta\eta} u
\end{aligned}$$

Observe, that we have used the β-rule in the first and the η-rule in the second equation. What about ξ? We need ξ to show that the assignment of $\phi_\to$ is natural in Γ, which is an requirement for an adjunction (we also need to show this for $\phi_\times$). I refer to Neil's course on category theory for the details of what this means.

Knowing a little bit category theory also reveals that I have been cheating: I really need to construct products for any two contexts and not just a context and a type and I have to define exponentials for contexts and show an appropriate isomorphism on substitutions and not just on terms. I leave it is an exercise to fix my cheats, but I's also like to add that my cheating isn't so bad and can be fixed by defining what *contextual* cartesian closed category (conCCC) is. Can't think about a good reference for that, though.

2.4 Combinatory logic

One of the main complications of λ-calculus is the presence of bound variables, which forces us to introduce α-equality and to generate fresh variables during substitutions. If we are only interested in β-equality we don't need λ but it is sufficient to introduce two combinators: K and S — we define $\mathrm{Tm}^{\mathrm{sk}}_{\Gamma}\,\sigma$ by omitting λ but instead introducing:

$$\frac{}{\mathrm{K}_{\sigma\tau} \in \mathrm{Tm}^{\mathrm{sk}}_{\Gamma}\,\sigma \to \tau \to \sigma} \qquad \frac{}{\mathrm{S}_{\sigma\tau\rho} \in \mathrm{Tm}^{\mathrm{sk}}_{\Gamma}\,(\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho}$$

We define $=_{\mathrm{sk}}$ by replacing the β-rule by

$$\frac{}{\mathrm{K}\,t\,u =_{\mathrm{sk}} t} \qquad \frac{}{\mathrm{S}\,t\,u\,v =_{\mathrm{sk}} t\,v\,(u\,v)}$$

If one sees the combinators the first time, one wonders what justifies this particular choice, especially of the complicated looking S. This is best understood by looking how we can define a derived lambda abstraction λ^*. First we notice that we can define an identity combinator $\mathrm{I}_\sigma = \mathrm{S}\,\mathrm{K}\,\mathrm{K}$ such that $\mathrm{I}\,t =_{\mathrm{sk}} t$. Now given $t \in \mathrm{Tm}^{\mathrm{sk}}_{\Gamma x:\sigma}\,\tau$ we define $\lambda^* x^\sigma.t \in \mathrm{Tm}^{\mathrm{sk}}_{\Gamma}\,\sigma \to \tau$ by recursion over t:

$$\begin{aligned} \lambda^* x.y &= \begin{cases} \mathrm{I} & \text{, if } x = y \\ \mathrm{K}\,y & \text{, if } x \neq y \end{cases} \\ \lambda^* x.t\,u &= \mathrm{S}\,(\lambda^* x.t)\,(\lambda^* x.u) \end{aligned}$$

Along with this definition we can establish that the β-axiom is a derived equation $(\lambda^* x.t)\,u =_{\mathrm{sk}} t[x = u]$. We can now define a translation assigning to a λ-term $t \in \mathrm{Tm}_\Gamma\,\sigma$ a combinator term $t^* \in \mathrm{Tm}^{\mathrm{sk}}_{\Gamma}\,\sigma$ by

$$\begin{aligned} x^* &= x \\ (\lambda x.t)^* &= \lambda^* x.t^* \\ (t\,u)^* &= t^*\,u^* \end{aligned}$$

Since the β-axiom is derivable, it is clear that we have that $t^* =_{\mathrm{sk}} u^*$, if $t =_\beta u$. It is also easy to define a translation in the other direction: given $t^* \in \mathrm{Tm}^{\mathrm{sk}}_{\Gamma}\,\sigma$ we define $t^\# \in \mathrm{Tm}_\Gamma\,\sigma$ by

$$\begin{aligned} \mathrm{K}^\# &= \lambda xy.y \\ \mathrm{S}^\# &= \lambda xyz.x\,z\,(y\,z) \end{aligned}$$

It is not hard to show that $(t^*)^\# =_\beta t$, but interestingly $(u^\#)^* =_{\mathrm{sk}} u$ fails, e.g. $(\mathrm{K}^\#)^* = \lambda^* xy.y = \mathrm{S}\,(\mathrm{K}\,\mathrm{K})\,\mathrm{I}$ but $\mathrm{K} \neq_{\mathrm{sk}} \mathrm{S}\,(\mathrm{K}\,\mathrm{K})\,\mathrm{I}$. Maybe the reader has an idea how to fix this (without loosing any of the other properties)?!

2.5 Functional encodings

The only types so far are type variables and function spaces. Much of the literature on the simply typed λ calculus concentrates on this situation, usually considering only the case of one type variable or base type, which for historical reasons is called $\circ$, i.e. $\mathcal{X} = \{\circ\}$. Datatypes like Bool and Nat then receive a functional encoding:

$$\begin{aligned}\mathrm{Bool} &= \circ \to \circ \to \circ\\ \mathrm{Nat} &= \circ \to (\circ \to \circ) \to \circ\end{aligned}$$

we can certainly define constructors for these encodings, for Bool

$$\begin{aligned}\mathrm{true} &\in \mathrm{Tm}\,\mathrm{Bool}\\ \mathrm{true} &= \lambda \mathtt{x}^{\circ}\lambda \mathtt{y}^{\circ}.\mathtt{x}\\ \mathrm{false} &\in \mathrm{Tm}\,\mathrm{Bool}\\ \mathrm{false} &= \lambda \mathtt{x}^{\circ}\lambda \mathtt{y}^{\circ}.\mathtt{y}\end{aligned}$$

and for Nat

$$\begin{aligned}\mathrm{zero} &\in \mathrm{Tm}\,\mathrm{Nat}\\ \mathrm{zero} &= \lambda \mathtt{z}^{\circ}.\lambda \mathtt{s}^{\circ\to\circ}.\mathtt{z}\\ \mathrm{succ} &\in \mathrm{Tm}\,\mathrm{Nat} \to \mathrm{Nat}\\ \mathrm{succ} &= \lambda \mathtt{n}^{\mathrm{Nat}}.\lambda \mathtt{z}^{\circ}.\lambda t^{\,\circ\to\circ}.\mathtt{s}\,(\mathtt{n}\,\mathtt{z}\,\mathtt{s})\end{aligned}$$

However, what can we do with the encoded data? In the case of Bool it is actually possible to define $\mathrm{if}^{\sigma} \in \mathrm{Bool} \to \sigma \to \sigma \to \sigma$ such that $\mathrm{if}\,\mathrm{true}\,t\,u =_{\beta\eta} t$ and $\mathrm{if}\,\mathrm{false}\,t\,u =_{\beta\eta} u$. However, the definition of if^{σ} (which I leave as an exercise) is not uniform in σ but proceeds by induction over σ and indeed only works if there is only one base type.

Even making these simplifying assumptions the situation for Nat is restrictive: only the extended polynomials are definable, this are the polynomials, step functions and their compositions (Exercise: define addition and multiplication). E.g. not even the predecessor is definable.

Conceptually, the encodings have a strange status: we have said that the idea of $\beta\eta$-equality is that functions are black boxes. However, to be able to differentiate between different encoded values such as true and false we have to look inside functions! Consequently, functions defined on encoded data are not extensional wrt. the encoded data.

Moreover, it doesn't seem to be the case that the encodings deliver a conceptual reduction: it seems easier to understand Bool or Nat as primitive notions that the idea of higher order functions over an uninterpreted base type.

In System F the encodings are more usable, becuase we can define all functions which are provable total in 2nd order logic. However, the other criticisms (not extensional, no conceptual reduction) remain valid. We can also add that a naive implementation of encoded types is inefficient and optimisation is not so easy.

2.6 $\lambda^{\to 2}$

Since I have argued that the functional encodings are no good, let's introduce some proper datatypes starting with

$$\frac{}{\mathrm{Bool} \in \mathrm{Ty}}$$

with new term formers

$$\frac{}{\mathrm{true}, \mathrm{false} \in \mathrm{Tm}_\Gamma\, \mathrm{Bool}} \qquad \frac{t \in \mathrm{Tm}_\Gamma\, \mathrm{Bool} \quad u_0, u_1 \in \mathrm{Tm}_\Gamma\, \sigma}{\mathrm{if}\, t\, u_0\, u_1 \in \mathrm{Tm}_\Gamma\, \sigma}$$

and the β-equations

$$\frac{}{\mathrm{if}\,\mathrm{true}\, u_0\, u_1 =_\beta u_0} \qquad \frac{}{\mathrm{if}\,\mathrm{false}\, u_0\, u_1 =_\beta u_1} \qquad \frac{t =_\beta t' \quad u_0 =_\beta u_0' \quad u_1 =_\beta u_1'}{\mathrm{if}\, t\, u_0\, u_1 =_\beta \mathrm{if}\, t'\, u_0'\, u_1'}$$

The $\beta\eta$-theory is a bit more involved. Basically we want to express that given $f \in \mathrm{Tm}_\Gamma\, \mathrm{Bool} \to \sigma$ if $f\, \mathrm{true} =_{\beta\eta} u_0$ and $f\, \mathrm{false} =_{\beta\eta} u_1$ then $f =_{\beta\eta} \lambda x.\mathrm{if}\, x\, u_0\, u_1$ for a fresh x. It turns our that we can avoid conditional equations here, but instead add

$$\frac{}{\mathrm{if}\, t\, \mathrm{true}\, \mathrm{false} =_{\beta\eta} t} \qquad \frac{}{t\, (\mathrm{if}\, u\, u_0\, u_1) =_{\beta\eta} \mathrm{if}\, u\, (t\, u_0)\, (t\, u_1)}$$

We can also view $\lambda^{\to 2}$ categorically: they correspond to a contextual CCC with a boolean object:

$$\phi_{\mathrm{Bool}} \in \mathrm{Tm}_{\Gamma \times \mathrm{Bool}}\, \sigma \cong (\mathrm{Tm}_\Gamma\, \sigma) \times (\mathrm{Tm}_\Gamma\, \sigma)$$

The equational theory introduces some interesting equalities. E.g., consider

$$\begin{aligned} \mathrm{once} &= \lambda f^{\mathrm{Bool}\to\mathrm{Bool}} \lambda x^{\mathrm{Bool}} f\, x \\ \mathrm{thrice} &= \lambda f^{\mathrm{Bool}\to\mathrm{Bool}} \lambda x^{\mathrm{Bool}} f\, (f\, (f\, x)) \end{aligned}$$

We observe that once $=_{\beta\eta}$ thrice. To see this, we note that, given $f : \mathrm{Bool} \to \mathrm{Bool}$, we have

$$\begin{aligned} f\, (f\, (f\, \mathrm{true})) &=_{\beta\eta} \mathrm{if}\, (f\, \mathrm{true})\, (f\, (f\, \mathrm{true}))\, (f\, (f\, \mathrm{false})) \\ &=_{\beta\eta} \mathrm{if}\, (f\, \mathrm{true})\, \mathrm{true}\, (f\, (f\, \mathrm{false})) \\ &=_{\beta\eta} \mathrm{if}\, (f\, \mathrm{true})\, \mathrm{true}\, (\mathrm{if}\, (f\, \mathrm{false})\, (f\, \mathrm{true})\, (f\, \mathrm{false})) \\ &=_{\beta\eta} \mathrm{if}\, (f\, \mathrm{true})\, \mathrm{true}\, (\mathrm{if}\, (f\, \mathrm{false})\, \mathrm{false}\, \mathrm{false}) \\ &=_{\beta\eta} \mathrm{if}\, (f\, \mathrm{true})\, \mathrm{true}\, \mathrm{false} \\ &=_{\beta\eta} f\, \mathrm{true} \end{aligned}$$

Symmetrically, we can show that $f\,(f\,(f\,\mathrm{false})) =_{\beta\eta} f\,\mathrm{false}$, and hence

$$\begin{array}{ll} \multicolumn{2}{l}{\mathrm{thrice}} \\ = & \lambda f^{\mathrm{Bool}\to\mathrm{Bool}}\lambda x^{\mathrm{Bool}} f\,(f\,(f\,x)) \\ =_{\beta\eta} & \lambda f^{\mathrm{Bool}\to\mathrm{Bool}}\lambda x^{\mathrm{Bool}}\mathrm{if}\,x\,(f\,(f\,(f\,\mathrm{true})))\,(f\,(f\,(f\,\mathrm{false}))) \\ =_{\beta\eta} & \lambda f^{\mathrm{Bool}\to\mathrm{Bool}}\lambda x^{\mathrm{Bool}}\mathrm{if}\,x\,(f\,\mathrm{true})\,(f\,\mathrm{false}) \\ =_{\beta\eta} & \lambda f^{\mathrm{Bool}\to\mathrm{Bool}}\lambda x^{\mathrm{Bool}} f\,x \\ = & \mathrm{once} \end{array}$$

It is easy to see that once and thrice are equal in the standard semantics where Bool is interpreted by a two-element set $\mathrm{Bool} = \{\mathrm{true}, \mathrm{false}\}$ and function types are set-theoretic function spaces. We observe that there are only four elements in $\mathrm{Bool} \to \mathrm{Bool} = \{\lambda x.\mathrm{true}, \lambda x.x, \lambda x.\neg x, \lambda x.\mathrm{false}\}$ and that for all the four $f \in \mathrm{Bool} \to \mathrm{Bool}$ we have $f^3 = f$. In [6] we show that this sort of reasoning always works.

2.7 $\lambda^{\to N}$

The natural numbers are, unlike Bool, an infinite type, which has the consequence that the extensional theory is no longer decidable. We introduce

$$\frac{}{\mathrm{Nat} \in \mathrm{Ty}}$$

with new term formers (here it stands for *iterator*):

$$\frac{}{\mathrm{zero} \in \mathrm{Tm}_\Gamma\,\mathrm{Nat}} \qquad \frac{n \in \mathrm{Tm}_\Gamma\,\mathrm{Nat}}{\mathrm{succ}\,n \in \mathrm{Tm}_\Gamma\,\mathrm{Nat}} \qquad \frac{n \in \mathrm{Tm}_\Gamma\,\mathrm{Nat} \quad z \in \mathrm{Tm}_\Gamma\,\sigma \quad s \in \mathrm{Tm}_\Gamma\,\sigma \to \sigma}{\mathrm{it}\,n\,z\,s \in \mathrm{Tm}_\Gamma\,\sigma}$$

and the β-equations

$$\frac{}{\mathrm{it}\,\mathrm{zero}\,z\,s =_\beta z} \qquad \frac{}{\mathrm{it}\,(\mathrm{succ}\,n)\,z\,s =_\beta s\,(\mathrm{it}\,n\,z\,s)} \qquad \frac{n =_\beta n' \quad z =_\beta z' \quad s =_\beta s'}{\mathrm{it}\,n\,z\,s =_\beta \mathrm{it}\,n\,z\,s}$$

$\beta\eta$-equality expresses the idea that two functions agree, if they agree on all constructors. Different than in the case for Bool we seem unable to avoid using a conditional equality:

$$\frac{}{\mathrm{it}\,n\,\mathrm{zero}\ \mathrm{succ} =_{\beta\eta} n} \qquad \frac{h\,(\mathrm{succ}\,i) =_{\beta\eta} f\,(h\,i)}{\mathrm{it}\,n\,(h\,\mathrm{zero})\,f =_{\beta\eta} h\,n}$$

For the fans of categories, I'd like to add that $\beta\eta$-equality corresponds to having a natural numbers object (NNO).

This theory is very strong, indeed undecidable. Just to give an example we can define addition

$$\mathrm{add} = \lambda m, n.\mathrm{it}\,m\,n\,\mathrm{succ}$$

and while we can use β-eqality to calculate:

$$\mathrm{add}\,(\mathrm{succ}^i\,\mathrm{zero})\,(\mathrm{succ}^j\,\mathrm{zero}) =_\beta \mathrm{succ}^{i+j}\,\mathrm{zero}$$

we can actually prove commutativity of addition using $\beta\eta$-equality:

$$\mathrm{it}\,m\,n\,\mathrm{succ} =_{\beta\eta} \mathrm{it}\,n\,m\,suc$$

Exercise: Derive this equality.

We can implement the proof that Peano arithmetic is undecidable in this theory, which is actually an equational version of arithmetic, that is it has the same provable equalities on Nat (I haven't checked this in detail).

An alternative to using is to introduce primitive recursion:

$$\frac{n \in \mathrm{Tm}_\Gamma\,\mathrm{Nat} \quad z \in \mathrm{Tm}_\Gamma\,\sigma \quad s \in \mathrm{Tm}_\Gamma\,\mathrm{Nat} \to \sigma \to \sigma}{\mathrm{prec}\,n\,z\,s \in \mathrm{Tm}_\Gamma\,\sigma}$$

with the β-equalities

$$\frac{}{\mathrm{prec}\,\mathrm{zero}\,z\,s =_\beta z} \qquad \frac{}{\mathrm{prec}\,(\mathrm{succ}\,n)\,z\,s =_\beta s\,n\,(\mathrm{it}\,n\,z\,s)} \qquad \frac{n =_\beta n' \quad z =_\beta z' \quad s =_\beta s'}{\mathrm{prec}\,n\,z\,s =_\beta \mathrm{prec}\,n\,z\,s}$$

In the presence of products we can define prec which is correct wrt. $\beta\eta$-equality. However, the encoding is not β-equal, which may be interpreted as implying that intensionally the iterator is insufficient. However, one may say the same about prec since there are other forms of recursion which intensionally cannot be implemented with prec, such as course-of-value recursion.

2.8 The untyped λ calculus

The untyped λ-calculus is the typed λ-calculus with no type variables but one type constant

$$\frac{}{\mathrm{Lam} \in \mathrm{Ty}}$$

and the equation $\mathrm{Lam} = \mathrm{Lam} \to \mathrm{Lam}$. If we don't want to have equations on types, we can alternatively add

$$\frac{f \in \mathrm{Tm}_\Gamma\,\mathrm{Lam} \to \mathrm{Lam}}{\mathrm{lam}\,f \in \mathrm{Tm}_\Gamma\,\mathrm{Lam}} \qquad \frac{d \in \mathrm{Tm}_\Gamma\,\mathrm{Lam}}{\mathrm{app}\,d \in \mathrm{Tm}_\Gamma\,\mathrm{Lam} \to \mathrm{Lam}}$$

and the equations

$$\frac{}{\mathrm{app}\,(\mathrm{lam}\,f) =_\beta f} \qquad \frac{}{\mathrm{lam}\,(\mathrm{app}\,d) =_{\beta\eta} d}$$

The untyped λ-calculus doesn't support the view that λ-terms are *total* functions. Untyped λ-terms correspond to computations, which may or may not terminate. We can show that precisely all Turing-computable functions are representable as untyped λ-terms using the encoding of natural numbers discussed in section 2.5.

3 Normalisation

Normalisation is a central theorem of typed λ-calculi giving rise to a canonical representation of terms upto β- or $\beta\eta$-equality. Together with confluence it entails decidability of equality but it also tells us that we are indeed taking about total functions. We may also use the induction over the canonical representations to establish further results.

The traditional approach to normalisation is based on a small-step reduction relation obtained by directing the rules defining β-equality (leaving out symmetry). There are a number of disadvantages of this naive approach: the small-step relation is not directly related to a reasonable implementation of reduction, one may say it is *how a mathematician would execute a program.* Also the small-step semantics is hard to reason about, even to show confluence, or Church-Rosser, requires some inguinuity, and it gets even worse when considering extensions of the basic theory. It seems that we first create a lot of trouble for ourselves and then have to work very hard to get out of it. Hence, I will present here two different approaches: the first one replaces the small-step reductions by a big-step reduction which can be easily implemented in a functional language such as Haskell. We could go further and turn this implementation into an abstract machine by making it tail-recursive. An alternative to using reduction is *normalisation by evaluation*, pioneered by Schwichtenberg and Berger [7] and further explored in recent research. The main idea is to invert the semantic interpretation function and thus obtain a syntactic representation for an equivalence class of terms. I will illustrate this for β-equality, e.g. see [8]. However, normalisation by evaluation assumes that you are already able to run functional programs on the metalevel, in particular it doesn't lead to a machine model to implement normalisation.

3.1 Normalisation by reduction for β-equality

Values are the result of our computation, proper values are λ-abstractions and neutral values are computations which got stuck because of free variables. In functional programming only the first case arises because all programs are closed. We define the set of values $\mathrm{Val}_\Gamma\,\sigma$ and neutral values $\mathrm{Ne}_\Gamma\,\sigma \subseteq \mathrm{Val}_\Gamma\,\sigma$ of type σ with free variables in Γ inductively

$$\frac{x \in \mathrm{Var}_\Gamma\,\sigma}{x \in \mathrm{Ne}_\Gamma\,\sigma} \qquad \frac{t \in \mathrm{Ne}_\Gamma\,\sigma \to \tau \quad v \in \mathrm{Val}_\Gamma\,\sigma}{t\,v \in \mathrm{Ne}_\Gamma\,\tau}$$

$$\frac{t \in \mathrm{Tm}_{\Gamma x^\sigma}\,\tau \quad \vec{v} \in \mathrm{Val}_\Delta\,\Gamma}{\lambda x^\sigma.t\{\vec{v}\} \in \mathrm{Val}_\Gamma\,\sigma}$$

Functional values are defined using closures, i.e. delayed substitutions. We can embed values into terms by carrying out those substitutions: given $v \in \mathrm{Val}_\Gamma\,\sigma$

or $v \in \mathrm{Ne}_\Gamma\,\sigma$ we define $\lceil v \rceil \in \mathrm{Tm}_\Gamma\,\sigma$ by

$$\begin{aligned} \lceil x \rceil &= x \\ \lceil t\,u \rceil &= \lceil t \rceil\,\lceil u \rceil \\ \lceil \lambda x^\sigma.t\{\vec{v}\} \rceil &= (\lambda x^\sigma.t)[\lceil \vec{v} \rceil] \end{aligned}$$

We inductively define a big-step reduction relation, and an auxilliary relation to reduce applications:

$$\frac{t \in \mathrm{Tm}_\Gamma\,\sigma \quad \vec{v} \in \mathrm{Val}_\Delta\,\Gamma}{t\{\vec{v}\} \Downarrow v\,\mathbf{Prop}} \qquad \frac{f \in \mathrm{Val}_\Gamma\,\sigma \to \tau \quad v \in \mathrm{Val}_\Gamma\,\sigma \quad w \in \mathrm{Val}_\Gamma\,\tau}{f\,v \Downarrow w\,\mathbf{Prop}}$$

by

$$\frac{}{x\{\vec{v}\} \Downarrow x[\vec{v}]} \qquad \frac{t[\vec{v}] \Downarrow f \quad u\{\vec{v}\} \Downarrow v \quad f\,v \Downarrow w}{t\,u\{\vec{v}\} \Downarrow w} \qquad \frac{}{\lambda x^\sigma.t\{\vec{v}\} \Downarrow (\lambda x^\sigma.t)[\vec{v}]}$$

$$\frac{t[\vec{v}, x = v] \Downarrow w}{(\lambda x^\sigma.t\{\vec{v}\})\,v \Downarrow w} \qquad \frac{}{n\,v \Downarrow n\,v}$$

This bigstep reduction implements *call-by-value* evaluation, because arguments are evaluated before they are passed to a function. This is also called *eager evaluation* and is used in the functional programming language SML. An alternative is *call-by-name*, or *lazy evaluation* as implemented in Haskell (ok, this is not the whole story). Here we put closures in the environment which only get evaluated ,if we actually need them. As a consequence of the results we are going to prove in this section, there is no (extensional) difference between call-by-value and call-by-name for typed terms. However, since the definition of reduction itself makes no reference to types, it also works for untyped terms. And for those there is a difference: more programs terminate for call-by-name. An example is $(\lambda xy.y)\,\Omega$, where $\Omega = (\lambda x.xx)(\lambda x.xx)$ is the prototypical diverging term. The call-by-value evaluator gets stuck becuase it tries to evaluate Ω first, while the call-by-value one never has a look at it. And indeed, one can show call-by-name is as good as it gets.

Exercise: Define call-by-name evaluation and reprove the results of this section.

We observe some obvious properties of $\Downarrow$:

Lemma 3.1. *Reduction is deterministic:*

$$\frac{t\{\vec{v}\} \Downarrow v \quad t\{\vec{v}\} \Downarrow v'}{v = v'} \qquad \frac{f\,v \Downarrow w \quad f\,v \Downarrow w'}{w = w'}$$

Proof. By induction over the derivation of $t\{\vec{v}\} \Downarrow v$ and $f\ v \Downarrow w$ and case analysis of the 2nd premise.

Lemma 3.2. *Reduction is sound, wrt.* $=_\beta$*:*

$$\frac{t\{\vec{v}\} \Downarrow v}{t[\lceil \vec{v} \rceil] =_\beta \lceil v \rceil}$$

Proof. Easy induction over the derivation of $t\{\vec{v}\} \Downarrow v$.

To show that this reduction relation can be used to decide β-equality we have to establish confluence and normalisation. To show confluence we need a lemma about substitution:

Lemma 3.3. $\dfrac{u\{\vec{v}\} \Downarrow v \quad t\{\vec{v}, x = v\} \Downarrow w \quad t[x = u]\{\vec{v}\} \Downarrow w'}{w = w'}$

Proof. By induction over $t \in \mathrm{Tm}_{\Gamma.x^\sigma}\, \tau$ using properties of substitution.

Proposition 3.4 (Confluence). *β-equal terms reduce to identical normal forms:*

$$\frac{t =_\beta t' \quad t\{\vec{v}\} \Downarrow v \quad t'\{\vec{v}\} \Downarrow v'}{v = v'} \qquad \frac{f =_\beta f' \quad v =_\beta v' \quad f\, v \Downarrow w \quad v'\, v' \Downarrow w'}{w = w'}$$

Proof. By induction over $t =_\beta t'$. The cases for (refl) and (sym) are obvious and (trans) follows from lemma 3.1. The interesting case is (β), here we need lemma 3.3.

We say that a term t is normalizing, if it reduces to a value. We write $t\{\vec{v}\} \Downarrow$ for this (to be precise $t\{\vec{v}\} \Downarrow$, iff $\exists v.t\{\vec{v}\} \Downarrow v$). First we observe that all values are normalizing (actually they normalize to themselves):

Lemma 3.5. *All values reduce are normalizing.*

$$\frac{\lceil v \rceil \in \mathrm{Val}_\Gamma\, \sigma}{\lceil v \rceil\{\vec{v}\} \Downarrow}$$

Proof. By induction over $v \in \mathrm{Val}_\Gamma\, \sigma$.

To show normalisation we have to load our induction and establish a stronger result, this is basically what is tradionally known as *reducibility.* We define a *logical predicate* $\mathrm{Red}^\sigma_\Gamma \subseteq \mathrm{Tm}_\Gamma\, \sigma$ (and $\mathrm{Red}^\Delta_G \subseteq \mathrm{Tm}_\Gamma\, \Delta$) by recursion on σ:

$$\begin{aligned}
\mathrm{Red}^X_\Gamma\, t &= \forall \vec{v} \in \mathrm{Val}_\Delta\, \Gamma.t\{\vec{v}\} \Downarrow \\
\mathrm{Red}^{\sigma\to\tau}_\Gamma\, t &= \quad \forall \vec{v} \in \mathrm{Val}_\Delta\, \Gamma.t\{\vec{v}\} \Downarrow \\
&\qquad \wedge \forall u \in \mathrm{Tm}_\Gamma\, \sigma.\mathrm{Red}^\sigma_\Gamma\, u \to \mathrm{Red}^\tau_\Gamma\, (t\, u) \\
\mathrm{Red}^{()}_\Gamma\, () &= \top \\
\mathrm{Red}^{\Delta x^\sigma}_\Gamma\, (\vec{u}, x = u) &= \mathrm{Red}^\Delta_\Gamma\, \vec{u} \wedge \mathrm{Red}^\sigma_\Gamma\, u
\end{aligned}$$

Note that reducible implies normalizing:

Lemma 3.6.

$$\frac{\mathrm{Red}^\sigma\, t \quad \vec{v} \in \mathrm{Val}_\Delta\, \Gamma}{t\{\vec{v}\} \Downarrow}$$

Proof. It's bloody obvious, isn't it.

We show that all terms are reducible, this is called the fundamental theorem for logical predicates:

Proposition 3.7. $\dfrac{t \in \mathrm{Tm}_\Gamma\,\sigma \quad \vec{u} \in \mathrm{Tm}_\Delta\,\Gamma \quad \mathrm{Red}^\Gamma_\Delta\,\vec{u}}{\mathrm{Red}^\sigma_\Delta\,t[\vec{u}]}$

Proof. by induction over $t \in \mathrm{Tm}_\Gamma\,\sigma$.

Now given $t \in \mathrm{Tm}_\Gamma\,\sigma$ we will use that $\mathrm{Red}^\Gamma_\Gamma\,1_\Gamma$ and hence $\mathrm{Red}^\sigma_\Gamma\,t[1_\Gamma]$ which implies that t is normalizing.

To obtain $\mathrm{Red}^\Gamma_\Gamma\,1_\Gamma$, we show that all neutral values are reducible

Lemma 3.8.

$$\frac{n \in \mathrm{Ne}_\Gamma\,\sigma}{\mathrm{Red}^\sigma_\Gamma\,n} \qquad \frac{\vec{v} \in \mathrm{Ne}_\Gamma\,\Delta}{\mathrm{Red}^\Delta_\Gamma\,v}$$

Proof. By induction over σ and Δ using lemma 3.5.

Proposition 3.9 (Normalisation). *Every term reduces to a value.*

$$\frac{t \in \mathrm{Tm}_\Gamma\,\sigma \quad \vec{v} \in \mathrm{Val}_\Delta\,\Gamma}{t\{\vec{v}\} \Downarrow}$$

Proof. By lemma 3.8 the identity substitution is reducible ($\mathrm{Red}\,1_\Gamma$). Hence by the fundamental theorem 3.7 we know $\mathrm{Red}\,t$ (since $t = t[1]$) and by lemma 3.6 $t\{\vec{v}\} \Downarrow$.

Corollary 3.10. $=_\beta$ *is decidable.*

Proof. If we want to check whether two terms $t, u \in \mathrm{Tm}_\Gamma\,\sigma$ are β-equal we apply them to the identity environment $1_\Gamma \in \mathrm{Val}_\Gamma\,\Gamma$ and use the normalisation property to calculate their normalforms $t[1_\Gamma] \Downarrow v, u[1_\Gamma] \Downarrow w$. Because of soundness (lemma 3.2) we know that $t =_\beta v, u =_\beta w$ and using confluence we can see that $v = w$ if and only if $t =_\beta u$.

Exercise: Extend the reduction relation and normalisation and confluence to $\lambda^{\to 2}$ *and* $\lambda^{\to N}$.

3.2 Normalisation by evaluation for β-equality

The idea is that we define a semantics for the calculus, i.e. an interpretation of typed and terms such that $t \in \mathrm{Tm}\,\sigma$ implies $[\![t]\!] \in [\![\sigma]\!]$ which is sound for β-equality ($t =_\beta t'$ implies $[\![t]\!] = [\![t']\!]$) and for which can effectively invert evaluation by a function $\mathrm{quote}^\sigma \in [\![\sigma]\!] \to \mathrm{Val}\,\sigma$ (such that $\mathrm{quote}^\sigma [\![t]\!] =_\beta t$)

The definition of values closely resembles the one in the previous section, with the only difference that we don't use closures:

$$\frac{x \in \mathrm{Var}_\Gamma\,\sigma}{x \in \mathrm{Ne}_\Gamma\,\sigma} \qquad \frac{t \in \mathrm{Ne}_\Gamma\,\sigma \to \tau \quad v \in \mathrm{Val}_\Gamma\,\sigma}{t\,v \in \mathrm{Ne}_\Gamma\,\tau}$$

$$\frac{t \in \mathrm{Ne}_\Gamma\,\sigma}{t \in \mathrm{Val}_\Gamma\,\sigma} \qquad \frac{t \in \mathrm{Tm}_{\Gamma x^\sigma}\,\tau}{\lambda x^\sigma.t \in \mathrm{Val}_\Gamma\,\sigma}$$

We proceed by defining an interpretation of types together with a function quote which extracts a value from a semantic object.

$$\begin{aligned}
[\![X]\!]_\Gamma &= \mathrm{Ne}_\Gamma\,X \\
\mathrm{quote}^X_\Gamma\,t = t & \\
[\![\sigma \to \tau]\!]_\Gamma &= \{(v \in \mathrm{Val}_\Gamma\,\sigma \to \tau, f \in [\![\sigma]\!]_\Gamma \to [\![\tau]\!]_\Gamma) \\
&\quad \mid \forall x \in [\![\sigma]\!]_\Gamma.\mathrm{quote}^\tau_\Gamma\,(f\,x) =_\beta v\,(\mathrm{quote}^\sigma_\Gamma\,x)\} \\
\mathrm{quote}^{\sigma\to\tau}_\Gamma\,(v,f) &= v
\end{aligned}$$

Given $(v,f) \in [\![\sigma \to \tau]\!]$ and $x \in [\![\sigma]\!]$ we write $(v,f)\,x = f\,x$ for semantic application.

The interpretation of terms

$$\frac{t \in \mathrm{Tm}_\Gamma\,\sigma \quad \vec{v} \in [\![\Gamma]\!]_\Delta}{[\![t]\!]\,\vec{v} \in [\![\sigma]\!]_\Delta}$$

is given by

$$\begin{aligned}
[\![x]\!]\,\vec{v} &= \vec{v}\,x \\
[\![t\,u]\!]\,\vec{v} &= ([\![t]\!]\,\vec{v})\,([\![u]\!]\,\vec{v}) \\
[\![\lambda x^\sigma.t]\!]\,\vec{v} &= (\lambda x^\sigma.t, v \in [\![\sigma]\!] \mapsto [\![t]\!]\,(\vec{x}, x = v))
\end{aligned}$$

It is easy to see that the semantics is closed under substitution:

Lemma 3.11.

$$[\![t]\!]\,(\vec{v}, x = [\![u]\!]\,\vec{v}) = [\![t[x = u]]\!]\,\vec{v}$$

Proof. By induction over $t \in \mathrm{Tm}_{\Gamma.x^\sigma}\,\tau$.

which is neded to show equational soundness

Proposition 3.12.

$$\frac{t =_\beta t'}{[\![t]\!] =_\beta [\![t']\!]}$$

Proof. By induction over $t =_\beta t'$. The only interesting case is (β) for which we use lemma 3.11.

It remains to show that quote actually inverts evaluation. We will use basically the same technique as in the last section, however, this time it is a logical relation instead of a logical predicate. We define

$$\frac{t \in \mathrm{Tm}_\Gamma\,\sigma \quad v \in [\![\sigma]\!]_\Gamma}{t R^\sigma_\Gamma v \in \mathbf{Prop}} \qquad \frac{\vec{t} \in \mathrm{Tm}_\Gamma\,\Delta \quad \vec{v} \in [\![\Delta]\!]_\Gamma}{\vec{t} R^\Delta_\Gamma \vec{v} \in \mathbf{Prop}}$$

by

$$\begin{aligned}
t R^X u &= t =_\beta u \\
t R_\Gamma^{\sigma\to\tau} f &= \forall u \in \mathrm{Tm}_\Gamma\, \sigma, v \in [\![\sigma]\!]_\Gamma . u R_\Gamma^\sigma v \implies t\, u R_\Gamma^\tau f\, v \\
() R^{()} () &= \top \\
(\vec{t}, x = t) R^{\Gamma . x^\sigma} (\vec{v}, x = v) &= \vec{t} R^\Gamma \vec{v} \wedge t R^\sigma v
\end{aligned}$$

As before for logical predicates (prop. 3.7) we establish a fundamental theorem:

Proposition 3.13. $\dfrac{t \in \mathrm{Tm}_\Gamma\, \sigma \quad \vec{t} R_\Delta^\Gamma \vec{v}}{t[\vec{t}] R_\Delta^\sigma [\![t]\!]\, \vec{v}}$

Proof. by induction over $t \in \mathrm{Tm}_\Gamma\, \sigma$.

The naming already indicates that both proposition 3.7 and 3.13 are instances of a more generic theorem. To formulate and prove this we need a notion of a semantical interpretation, the fact that thise are closed under products and the generic fundamental theorem for any interpretation.

To establish our main result we show

Lemma 3.14. *If $t R^\sigma v$ then $t = \mathrm{quote}^\sigma v$.*

Proof. by induction over σ.

To be able to show directly that normalisation extends to open term we need a lemma on neutral terms:

Lemma 3.15.

$$\frac{n \in \mathrm{Ne}_\Gamma\, \sigma}{n R^\sigma [\![n]\!]} \qquad \frac{\vec{n} \in \mathrm{Ne}_\Gamma\, \Delta}{\vec{n} R^\Delta [\![\vec{n}]\!]}$$

Proof. By induction over σ and Δ using lemma 3.14

We can now define a normalisation function:

$$\frac{t \in \mathrm{Tm}_\Gamma\, \sigma}{\mathrm{nf}\, t \in \mathrm{Val}_\Gamma\, \sigma}$$

by

$$\mathrm{nf}\, t = \mathrm{quote}\, ([\![t]\!]\, 1_\Gamma)$$

To see that this is indeed a normalisation function we observe that by lemma 3.12 $t =_\beta u$ implies $\mathrm{nf}\, t = \mathrm{nf}\, u$. Furthermore exploiting lemmas 3.15, 3.14 and proposition 3.13 we can deduce that $t =_\beta \mathrm{nf}\, t$.

Exercise: Extend normalisation by evaluation for the β-equalities of $\lambda^{\to 2}$ and $\lambda^{\to N}$.

4 Going further

Due to time constraints I had to leave out some material on Normalisation by evaluation for $\beta\eta$-equality, simplifying the account published in [5]. We need a different notion of logical relations, called Kripke logical relations, see [2].

This method also extends to products ($\times$) but coproducts ($+$) are a harder nut. In [4] we used concepts from sheaf theory to solve this problem. In the special case of $\lambda^{\rightarrow 2}$ with no type variable there is a simple construction which needs to a nice implementation, see [6].

Once we have got products and coproducts, we can add inductive (μ) and coinductive types (ν) , e.g. $\mathrm{Nat} = \mu X.1 + X$ or streams $\mathrm{Stream}\, A = \nu X.A \times X$. If we want to preserve normalisation, we have to introduce some constraints on μ and ν types to rule out types like $\mu X.X \rightarrow X$ which encodes the untyped λ-calculus. We can forbid $\rightarrow$ within μ, which, omitting ν-types, leads to a system, which can be encoded within $\lambda^{\rightarrow N}$ (I call this glorified arithmetic). A more general choice are strictly positive types such as infinitely branching tree $\mu X.1 + \mathrm{Nat} \rightarrow X$. Even less restrictive are positive types such as $\mu X.(X \rightarrow \mathrm{Bool}) \rightarrow \mathrm{Bool}$. However they have no set-theoretic model and the normalisation proof requires impredicative reasoning. Indeed, we can encode all positive inductive and coinductive types in System F — this encoding is sound for β. To obtain soundness for the $\beta\eta$-equality we need parametricity in System F.

[1] introduces $\lambda^{\mu\nu}$ and shows strong normalisation for a small step semantics. However, there shouldn't be any problem to extend the technique presented here to this situation.

Dependent types add some technical complications, because types and terms are mutually dependent. However, the benefits in expressivity are ernomous, we can completely formalize constructive set-theoretic reasoning within such a system. To get a taste of this, I refer to Conor's course.

References

1. A. Abel and T. Altenkirch. A predicative strong normalisation proof for a λ-calculus with interleaving inductive types. In *Types for Proof and Programs, International Workshop, TYPES '99, Selected Papers*, volume 1956 of *Lecture Notes in Computer Science*, 2000.
2. T. Altenkirch. Notes on definability and Kripke logical relations. available online, 2000.
3. T. Altenkirch. α-conversion is easy. Under Revision, 2002.
4. T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, 2001.
5. T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.
6. T. Altenkirch and T. Uustalu. Normalization by evaluation for $\lambda^{\rightarrow 2}$. In *Functional and Logic Programming*, 2004.
7. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proc. of 6th Annual IEEE Symposium on Logic in Computer Science*, pages 202–211. IEEE CS Press, 1991.
8. T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.
9. B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.