A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family

Nils Anders Danielsson

Chalmers University of Technology

Abstract. It is demonstrated how a dependently typed lambda calculus (a logical framework) can be formalised inside a language with inductive-recursive families. The formalisation does not use raw terms; the well-typed terms are defined directly. It is hence impossible to create ill-typed terms.
As an example of programming with strong invariants, and to show that the formalisation is usable, normalisation is proved. Moreover, this proof seems to be the first formal account of normalisation by evaluation for a dependently typed language.

1 Introduction

Programs can be verified in many different ways. One difference lies in how invariants are handled. Consider a type checker, for instance. The typing rules of the language being type checked are important invariants of the resulting abstract syntax. In the *external* approach to handling invariants the type checker works with raw terms. Only later, when verifying the soundness of the type checker, is it necessary to verify that the resulting, supposedly well-typed terms satisfy the invariants (typing rules). In the *internal* approach the typing rules are instead represented directly in the abstract syntax data types, and soundness thus follows automatically from the type of the type checking function, possibly at the cost of extra work in the implementation. For complicated invariants the internal approach requires strong forms of data types, such as inductive families or generalised algebraic data types.

Various aspects of many different essentially simply typed programming languages have been formalised using the internal approach [CD97, AR99, Coq02, XCC03, PL04, MM04, AC06, MW06, McBb]. Little work has been done on formalising *dependently* typed languages using this approach, though; Dybjer's work [Dyb96] on formalising so-called categories with families, which can be seen as the basic framework of dependent types, seems to be the only exception. The present work attempts to fill this gap.

This paper describes a formalisation of the type system, and a proof of normalisation, for a dependently typed lambda calculus (basically the logical framework of Martin-Löf's monomorphic type theory [NPS90] with explicit substitutions). Moreover, the proof of normalisation seems to be the first formal

implementation of normalisation by evaluation [ML75, BS91] for a dependently typed language. The ultimate goal of this work is to have a formalised implementation of the core type checker for a full-scale implementation of type theory.

To summarise, the contributions of this work are as follows:

- A fully typed representation of a dependently typed language (Sect. 3).
- A proof of normalisation (Sect. 5). This proof seems to be the first account of a formal implementation of normalisation by evaluation for a dependently typed language.
- Everything is implemented and type checked in the proof checker AgdaLight [Nor07]. The code can be downloaded from the author's web page [Dan07].

2 Meta Language

Let us begin by introducing the meta language in which the formalisation has been carried out, AgdaLight [Nor07], a prototype of a dependently typed programming language. It is in many respects similar to Haskell, but, naturally, deviates in some ways.

One difference is that AgdaLight lacks polymorphism, but has *hidden arguments*, which in combination with dependent types compensate for this loss. For instance, the ordinary list function *map* could be given the following type signature:

$$map : \{a, b : Set\} \to (a \to b) \to List\ a \to List\ b$$

Here *Set* is the type of types from the first universe. Arguments within $\{\ldots\}$ are hidden, and need not be given explicitly, if the type checker can infer their values from the context in some way. If the hidden arguments cannot be inferred, then they can be given explicitly by enclosing them within $\{\ldots\}$:

$$map\ \{Integer\}\ \{Bool\} : (Integer \to Bool) \to List\ Integer \to List\ Bool$$

AgdaLight also has inductive-recursive families [DS06], illustrated by the following example (which is not recursive, just inductive). Data types are introduced by listing the constructors and giving their types; natural numbers can for instance be defined as follows:

$$\begin{array}{l}\textbf{data}\ Nat : Set\ \textbf{where}\\ \quad zero : Nat\\ \quad suc\ : Nat \to Nat\end{array}$$

Vectors, lists of a given fixed length, may be more interesting:

$$\begin{array}{l}\textbf{data}\ Vec\ (a : Set) : Nat \to Set\ \textbf{where}\\ \quad nil\ \ : Vec\ a\ zero\\ \quad cons : \{n : Nat\} \to a \to Vec\ a\ n \to Vec\ a\ (suc\ n)\end{array}$$

Note how the *index* (the natural number introduced after the last : in the definition of *Vec*) is allowed to vary between the constructors. *Vec a* is a *family* of types, with one type for every index n.

To illustrate the kind of pattern matching AgdaLight allows for an inductive family, let us define the tail function:

$$\begin{aligned}&\mathit{tail} : \{\, a : \mathit{Set} \,\} \;\to\; \{\, n : \mathit{Nat} \,\} \;\to\; \mathit{Vec}\ a\ (\mathit{suc}\ n) \;\to\; \mathit{Vec}\ a\ n\\ &\mathit{tail}\ (\mathit{cons}\ x\ \mathit{xs}) = \mathit{xs}\end{aligned}$$

We can and need only pattern match on *cons*, since the type of *nil* does not match the type *Vec a* (*suc n*) given in the type signature for *tail*. As another example, consider the definition of the append function:

$$\begin{aligned}&(+\!\!+) : \mathit{Vec}\ a\ n_1 \;\to\; \mathit{Vec}\ a\ n_2 \;\to\; \mathit{Vec}\ a\ (n_1 + n_2)\\ &\mathit{nil} \qquad +\!\!+\ \mathit{ys} = \mathit{ys}\\ &\mathit{cons}\ x\ \mathit{xs} +\!\!+\ \mathit{ys} = \mathit{cons}\ x\ (\mathit{xs} +\!\!+\ \mathit{ys})\end{aligned}$$

In the *nil* case the variable n_1 in the type signature is unified with *zero*, transforming the result type into *Vec a* n_2, allowing us to give *ys* as the right-hand side. (This assumes that *zero* + n_2 evaluates to n_2.) The *cons* case can be explained in a similar way.

Note that the hidden arguments of (++) were not declared in its type signature. This is not allowed by AgdaLight, but often done in the paper to reduce notational noise. Some other details of the formalisation are also ignored, to make the paper easier to follow. The actual code can be downloaded for inspection [Dan07].

Note also that some of the inductive-recursive families in this formalisation do not quite meet the requirements of [DS06]; see Sects. 3.2 and 5.2. Furthermore [DS06] only deals with functions defined using elimination rules. The functions in this paper are defined using pattern matching and structural recursion.

AgdaLight currently lacks (working) facilities for checking that the code is terminating and that all pattern matching definitions are exhaustive. However, for the formalisation presented here this has been verified manually. Unless some mistake has been made all data types are strictly positive (with the exception of *Val*; see Sect. 5.2), all definitions are exhaustive, and every function uses structural recursion of the kind accepted by the termination checker foetus [AA02].

3 Object Language

The object language that is formalised is a simple dependently typed lambda calculus with explicit substitutions. Its type system is sketched in Fig. 1. The labels on the rules correspond to constructors introduced in the formalisation. Note that $\Gamma \Rightarrow \Delta$ is the type of substitutions taking terms with variables in Γ to terms with variables in Δ, and that the symbol $=_\star$ stands for $\beta\eta$-equality between types. Some things are worth noting about the language:

Contexts

$$\frac{}{\varepsilon\ context}\ (\varepsilon) \qquad \frac{\Gamma\ context \qquad \Gamma \vdash \tau\ type}{\Gamma, x : \tau\ context}\ (\triangleright)$$

Types

$$\frac{}{\Gamma \vdash \star\ type}\ (\star) \qquad \frac{\Gamma \vdash \tau_1\ type \qquad \Gamma, x : \tau_1 \vdash \tau_2\ type}{\Gamma \vdash \Pi(x : \tau_1)\,\tau_2\ type}\ (\Pi) \qquad \frac{\Gamma \vdash t : \star}{\Gamma \vdash El\ t\ type}\ (El)$$

Terms

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ (var) \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \Pi(x : \tau_1)\,\tau_2}\ (\lambda) \qquad \frac{\Gamma \vdash t : \tau \qquad \rho\ :\ \Gamma \Rightarrow \Delta}{\Delta \vdash t\ \rho : \tau\ \rho}\ (/\!\vdash)$$

$$\frac{\Gamma \vdash t_1 : \Pi(x : \tau_1)\,\tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2\ [x \mapsto t_2]}\ (@) \qquad \frac{\Gamma \vdash t : \tau_1 \qquad \tau_1 =_\star \tau_2}{\Gamma \vdash t : \tau_2}\ (::^{\equiv}_{\vdash})$$

Substitutions

$$\frac{\Gamma \vdash t : \tau}{[x \mapsto t]\ :\ \Gamma, x : \tau \Rightarrow \Gamma}\ (sub) \qquad \frac{}{wk\ x\ \tau\ :\ \Gamma \Rightarrow \Gamma, x : \tau}\ (wk) \qquad \frac{}{id\ \Gamma\ :\ \Gamma \Rightarrow \Gamma}\ (id)$$

$$\frac{\rho\ :\ \Gamma \Rightarrow \Delta}{\rho \uparrow_x \tau\ :\ \Gamma, x : \tau \Rightarrow \Delta, x : \tau\ \rho}\ (\uparrow) \qquad \frac{\rho_1\ :\ \Gamma \Rightarrow \Delta \qquad \rho_2\ :\ \Delta \Rightarrow X}{\rho_1\ \rho_2\ :\ \Gamma \Rightarrow X}\ (\odot)$$

Fig. 1: Sketch of the type system that is formalised. If a rule mentions $\Gamma \vdash t : \tau$, then it is implicitly assumed that Γ *context* and $\Gamma \vdash \tau$ *type*; similar assumptions apply to the other judgements as well. All freshness side conditions have been omitted.

- It has explicit substitutions in the sense that the application of a substitution to a term is an explicit construction in the language. However, the application of a substitution to a *type* is an implicit operation.
- There does not seem to be a "standard" choice of basic substitutions. The set chosen here is the following:
  - $[x \mapsto t]$ is the substitution mapping x to t and every other variable to itself.
  - $wk\ x\ \tau$ extends the context with a new, unused variable.
  - $id\ \Gamma$ is the identity substitution on Γ.
  - $\rho \uparrow_x \tau$ is a lifting; variable x is mapped to itself, and the other variables are mapped by ρ.
  - $\rho_1\ \rho_2$ is composition of substitutions.
- Heterogeneous equality is used. Two types can be equal ($\tau_1 =_\star \tau_2$) even though their contexts are not definitionally equal in the meta-theory. Contexts of equal types are always provably equal in the object-theory, though (see Sect. 3.6).

The following subsections describe the various parts of the formalisation: contexts, types, terms, variables, substitutions and equalities. Section 3.7 discusses

some of the design choices made. The table below summarises the types defined; the concept being defined, typical variable names used for elements of the type, and the type name (fully applied):

Contexts	Γ, Δ, X	$Ctxt$
Types	τ, σ	$Ty\ \Gamma$
Terms	t	$\Gamma \vdash \tau$
Variables	v	$\Gamma \ni \tau$
Substitutions	ρ	$\Gamma \Rightarrow \Delta$
Equalities	eq	$\Gamma_1 =_{Ctxt} \Gamma_2,\ \tau_1 =_\star \tau_2, \ldots$

Note that all the types in this section are part of the same mutually recursive definition, together with the function $(/)$ (see Sect. 3.2).

3.1 Contexts

Contexts are represented in a straight-forward way. The empty context is written ε, and $\Gamma \triangleright \tau$ is the context Γ extended with the type τ. Variables are represented using de Bruijn indices, so there is no need to mention variables here:

$$\begin{aligned}&\textbf{data}\ Ctxt : Set\ \textbf{where}\\ &\quad \varepsilon\ \ \ : Ctxt\\ &\quad (\triangleright) : (\Gamma : Ctxt) \rightarrow Ty\ \Gamma \rightarrow Ctxt\end{aligned}$$

$Ty\ \Gamma$ is the type, introduced below, of object-language types with variables in Γ.

3.2 Types

The definition of the type family Ty of object-level types follows the type system sketch in Fig. 1:

$$\begin{aligned}&\textbf{data}\ Ty : Ctxt \rightarrow Set\ \textbf{where}\\ &\quad \star : \{\Gamma : Ctxt\} \rightarrow Ty\ \Gamma\\ &\quad \Pi : (\tau : Ty\ \Gamma) \rightarrow Ty\ (\Gamma \triangleright \tau) \rightarrow Ty\ \Gamma\\ &\quad El : \Gamma \vdash \star \rightarrow Ty\ \Gamma\end{aligned}$$

The type $\Gamma \vdash \tau$ stands for terms of type τ with variables in Γ, so terms can only be viewed as types if they have type $\star$.

Note that types are indexed on the context to which their variables belong, and similarly terms are indexed on both contexts and types ($\Gamma \vdash \tau$). The meta-theory behind indexing a type by a type family defined in the *same* mutually recursive definition has not been worked out properly yet. It is, however, crucial to this formalisation.

Let us now define the function $(/)$, which applies a substitution to a type (note that postfix application is used). The type $\Gamma \Rightarrow \Delta$ stands for a substitution which, when applied to something in context Γ (a type, for instance), transforms this into something in context Δ:

$$
\begin{aligned}
&(/) : \mathit{Ty}\ \Gamma \to \Gamma \Rightarrow \Delta \to \mathit{Ty}\ \Delta \\
&\star \quad\ \ / \rho = \star \\
&\Pi\ \tau_1\ \tau_2 / \rho = \Pi\ (\tau_1 / \rho)\ (\tau_2 / \rho \uparrow \tau_1) \\
&\mathit{El}\ t \quad / \rho = \mathit{El}\ (t \mathbin{/\!\vdash} \rho)
\end{aligned}
$$

The constructor $(/\!\vdash)$ is the analogue of $(/)$ for terms (see Sect. 3.3). The substitution transformer $(\uparrow)$ is used when going under binders; $\rho \uparrow \tau_1$ behaves as ρ, except that the new variable zero in the original context is mapped to the new variable zero in the resulting context:

$$(\uparrow) : (\rho : \Gamma \Rightarrow \Delta) \to (\sigma : \mathit{Ty}\ \Gamma) \to \Gamma \triangleright \sigma \Rightarrow \Delta \triangleright (\sigma / \rho)$$

Substitutions are defined in Sect. 3.5.

3.3 Terms

The types $\Gamma \vdash \tau$ and $\Gamma \ni \tau$ stand for terms and variables, respectively, of type τ in context Γ. Note that what is customarily written $\Gamma \vdash t : \tau$, like in Fig. 1, is now written $t : \Gamma \vdash \tau$. There are five kinds of terms: variables (*var*), abstractions (λ), applications (@), casts ($::^{\equiv}_{\vdash}$) and substitution applications ($/\!\vdash$):

$$
\begin{aligned}
&\mathbf{data}\ (\vdash) : (\Gamma : \mathit{Ctxt}) \to \mathit{Ty}\ \Gamma \to \mathit{Set}\ \mathbf{where} \\
&\quad \mathit{var}\ \ : \Gamma \ni \tau && && \to \Gamma \vdash \tau \\
&\quad \lambda\ \ \ : \Gamma \triangleright \tau_1 \vdash \tau_2 && && \to \Gamma \vdash \Pi\ \tau_1\ \tau_2 \\
&\quad (@)\ \ : \Gamma \vdash \Pi\ \tau_1\ \tau_2 && \to (t_2 : \Gamma \vdash \tau_1) && \to \Gamma \vdash \tau_2 / \mathit{sub}\ t_2 \\
&\quad (::^{\equiv}_{\vdash}) : \Gamma \vdash \tau_1 && \to \tau_1 =_\star \tau_2 && \to \Gamma \vdash \tau_2 \\
&\quad (/\!\vdash)\ \ : \Gamma \vdash \tau && \to (\rho : \Gamma \Rightarrow \Delta) && \to \Delta \vdash \tau / \rho
\end{aligned}
$$

Notice the similarity to the rules in Fig. 1. The substitution $\mathit{sub}\ t_2$ used in the definition of (@) replaces *vz* with t_2, and lowers the index of all other variables by one:

$$\mathit{sub} : \Gamma \vdash \tau \to \Gamma \triangleright \tau \Rightarrow \Gamma$$

The conversion rule defined here ($::^{\equiv}_{\vdash}$) requires the two contexts to be definitionally equal in the meta-theory. A more general version of the rule would lead to increased complexity when functions that pattern match on terms are defined. However, we can *prove* a general version of the conversion rule, so no generality is lost:

$$(::_{\vdash}) : \Gamma_1 \vdash \tau_1 \to \tau_1 =_\star \tau_2 \to \Gamma_2 \vdash \tau_2$$

In this formalisation, whenever a cast constructor named $(::^{\equiv}_{\bullet})$ is introduced (where $\bullet$ can be $\vdash$ or $\ni$, for instance), a corresponding generalised variant $(::_{\bullet})$ is always proved.

Before moving on to variables, note that all typing information is present in a term, including casts (the conversion rule). Hence this type family actually represents typing derivations.

3.4 Variables

Variables are represented using de Bruijn indices (the notation $(\ni)$ is taken from [McBb]):

$$
\begin{array}{l}
\textbf{data}\ (\ni) : (\Gamma : \mathit{Ctxt}) \to \mathit{Ty}\ \Gamma \to \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{vz} & : & \{\sigma : \mathit{Ty}\ \Gamma\} \to \Gamma \triangleright \sigma \ni \sigma \,/\, \mathit{wk}\ \sigma \\
\mathit{vs} & : \Gamma \ni \tau \to & \{\sigma : \mathit{Ty}\ \Gamma\} \to \Gamma \triangleright \sigma \ni \tau \,/\, \mathit{wk}\ \sigma \\
(::^{\equiv}_{\ni}) & : \Gamma \ni \tau_1 \to \tau_1 =_\star \tau_2 & \to \Gamma \ni \tau_2
\end{array}
\end{array}
$$

The rightmost variable in the context is denoted by vz ("variable zero"), and $\mathit{vs}\ v$ is the variable to the left of v. The substitution $\mathit{wk}\ \sigma$ is a weakening, taking something in context Γ to the context $\Gamma \triangleright \sigma$:

$$\mathit{wk} : (\sigma : \mathit{Ty}\ \Gamma) \to \Gamma \Rightarrow \Gamma \triangleright \sigma$$

The use of weakening is necessary since, for instance, σ is a type in Γ, whereas vz creates a variable in $\Gamma \triangleright \sigma$.

The constructor $(::^{\equiv}_{\ni})$ is a variant of the conversion rule for variables. It might seem strange that the conversion rule is introduced twice, once for variables and once for terms. However, note that $\mathit{var}\ v ::^{\equiv}_{\vdash} \mathit{eq}$ is a term and not a variable, so if the conversion rule is needed to show that a variable has a certain type, then $(::^{\equiv}_{\vdash})$ cannot be used.

3.5 Substitutions

Substitutions are defined as follows:

$$
\begin{array}{l}
\textbf{data}\ (\Rightarrow) : \mathit{Ctxt} \to \mathit{Ctxt} \to \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{sub} : \Gamma \vdash \tau & & \to \Gamma \triangleright \tau \Rightarrow \Gamma \\
\mathit{wk} : (\sigma : \mathit{Ty}\ \Gamma) & & \to \Gamma \Rightarrow \Gamma \triangleright \sigma \\
(\uparrow) : (\rho : \Gamma \Rightarrow \Delta) & \to (\sigma : \mathit{Ty}\ \Gamma) & \to \Gamma \triangleright \sigma \Rightarrow \Delta \triangleright (\sigma \,/\, \rho) \\
\mathit{id} : & & \Gamma \Rightarrow \Gamma \\
(\odot) : \Gamma \Rightarrow \Delta & \to \Delta \Rightarrow X & \to \Gamma \Rightarrow X
\end{array}
\end{array}
$$

Single-term substitutions (sub), weakenings (wk) and liftings ($\uparrow$) have been introduced above. The remaining constructors denote the identity substitution (id) and composition of substitutions ($\odot$). The reasons for using this particular definition of $(\Rightarrow)$ are outlined in Sect. 3.7.

3.6 Equality

The following equalities are defined:

$$
\begin{array}{lll}
(=_{\mathit{Ctxt}}) : \mathit{Ctxt} & \to \mathit{Ctxt} & \to \mathit{Set} \\
(=_\star) \quad : \mathit{Ty}\ \Gamma_1 & \to \mathit{Ty}\ \Gamma_2 & \to \mathit{Set} \\
(=_\vdash) \quad : \Gamma_1 \vdash \tau_1 & \to \Gamma_2 \vdash \tau_2 & \to \mathit{Set}
\end{array}
$$

$$
\begin{array}{ll}
(=_{\ni}) & : \Gamma_1 \ni \tau_1 \;\rightarrow\; \Gamma_2 \ni \tau_2 \;\rightarrow\; \mathit{Set} \\
(=_{\Rightarrow}) & : \Gamma_1 \Rightarrow \Delta_1 \;\rightarrow\; \Gamma_2 \Rightarrow \Delta_2 \;\rightarrow\; \mathit{Set}
\end{array}
$$

As mentioned above heterogeneous equality is used. As a sanity check every equality is associated with one or more lemmas like the following one, which states that equal terms have equal types:

$$
eq_{\vdash} eq_{\star} : \{\, t_1 : \Gamma_1 \vdash \tau_1 \,\} \;\rightarrow\; \{\, t_2 : \Gamma_2 \vdash \tau_2 \,\} \;\rightarrow\; t_1 =_{\vdash} t_2 \;\rightarrow\; \tau_1 =_{\star} \tau_2
$$

The context and type equalities are the obvious congruences. For instance, type equality is defined as follows:

$$
\begin{array}{l}
\textbf{data}\ (=_{\star}) : \mathit{Ty}\ \Gamma_1 \;\rightarrow\; \mathit{Ty}\ \Gamma_2 \;\rightarrow\; \mathit{Set}\ \textbf{where} \\
\quad \star_{Cong} \;: \Gamma_1 =_{Ctxt} \Gamma_2 \;\rightarrow\; \star\{\Gamma_1\} =_{\star} \star\{\Gamma_2\} \\
\quad \Pi_{Cong} \;: \tau_{11} =_{\star} \tau_{12} \;\rightarrow\; \tau_{21} =_{\star} \tau_{22} \;\rightarrow\; \Pi\ \tau_{11}\ \tau_{21} =_{\star} \Pi\ \tau_{12}\ \tau_{22} \\
\quad \mathit{El}_{Cong} : t_1 =_{\vdash} t_2 \;\rightarrow\; \mathit{El}\ t_1 =_{\star} \mathit{El}\ t_2
\end{array}
$$

In many presentations of type theory it is also postulated that type equality is an equivalence relation. This introduces an unnecessary amount of constructors into the data type; when proving something about a data type one typically needs to pattern match on all its constructors. Instead I have chosen to *prove* that every equality (except $(=_{\vdash})$) is an equivalence relation:

$$
\begin{array}{ll}
\mathit{refl}_{\star} & : \tau \;=_{\star} \tau \\
\mathit{sym}_{\star} & : \tau_1 =_{\star} \tau_2 \;\rightarrow\; \tau_2 =_{\star} \tau_1 \\
\mathit{trans}_{\star} & : \tau_1 =_{\star} \tau_2 \;\rightarrow\; \tau_2 =_{\star} \tau_3 \;\rightarrow\; \tau_1 =_{\star} \tau_3
\end{array}
$$

(And so on for the other equalities.)

The semantics of a variable should not change if a cast is added, so the variable equality is a little different. In order to still be able to prove that the relation is an equivalence the following definition is used:

$$
\begin{array}{l}
\textbf{data}\ (=_{\ni}) : \Gamma_1 \ni \tau_1 \;\rightarrow\; \Gamma_2 \ni \tau_2 \;\rightarrow\; \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lllll}
\mathit{vz}_{Cong} & : & \sigma_1 =_{\star} \sigma_2 \;\rightarrow\; \mathit{vz} & \{\sigma_1\} =_{\ni} \mathit{vz} & \{\sigma_2\} \\
\mathit{vs}_{Cong} & : v_1 =_{\ni} v_2 \;\rightarrow\; \sigma_1 =_{\star} \sigma_2 \;\rightarrow\; \mathit{vs}\ v_1\,\{\sigma_1\} & =_{\ni} \mathit{vs}\ v_2\,\{\sigma_2\} & \\
\mathit{castEq}^{\ell}_{\ni} & : v_1 =_{\ni} v_2 \;\rightarrow\; v_1 ::^{\equiv}_{\ni} \mathit{eq} & =_{\ni} v_2 & \\
\mathit{castEq}^{r}_{\ni} & : v_1 =_{\ni} v_2 \;\rightarrow\; v_1 & =_{\ni} v_2 ::^{\equiv}_{\ni} \mathit{eq} &
\end{array}
\end{array}
$$

For substitutions extensional equality is used:

$$
\begin{array}{l}
\textbf{data}\ (=_{\Rightarrow})\ (\rho_1 : \Gamma_1 \Rightarrow \Delta_1)\ (\rho_2 : \Gamma_2 \Rightarrow \Delta_2) : \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{extEq} & : & \Gamma_1 =_{Ctxt} \Gamma_2 \;\rightarrow\; \Delta_1 =_{Ctxt} \Delta_2 \\
 & & \;\rightarrow\; (\forall v_1\ v_2.\, v_1 =_{\ni} v_2 \;\rightarrow\; \mathit{var}\ v_1 \not\vdash \rho_1 =_{\vdash} \mathit{var}\ v_2 \not\vdash \rho_2) \\
 & & \;\rightarrow\; \rho_1 =_{\Rightarrow} \rho_2
\end{array}
\end{array}
$$

Note that this data type contains negative occurrences of Ty, $(\ni)$ and $(=_{\ni})$, which are defined in the same mutually recursive definition as $(=_{\Rightarrow})$. In order to keep this definition strictly positive a first-order variant of $(=_{\Rightarrow})$ is used, which

simulates the higher-order version by explicitly enumerating all the variables. The first-order variant is later proved equivalent to the definition given here.

Term equality is handled in another way than the other equalities. The presence of the β and η laws makes it hard to prove that $(=_\vdash)$ is an equivalence relation, and hence this is postulated:

$$
\begin{aligned}
&\textbf{data}\ (=_\vdash) : \Gamma_1 \vdash \tau_1 \to \Gamma_2 \vdash \tau_2 \to \mathit{Set}\ \textbf{where} \\
&\quad \text{-- Equivalence.} \\
&\quad \mathit{refl}_\vdash \quad : (t : \Gamma \vdash \tau) \to t =_\vdash t \\
&\quad \mathit{sym}_\vdash \quad : t_1 =_\vdash t_2 \to t_2 =_\vdash t_1 \\
&\quad \mathit{trans}_\vdash : t_1 =_\vdash t_2 \to t_2 =_\vdash t_3 \to t_1 =_\vdash t_3 \\
&\quad \text{-- Congruence.} \\
&\quad \mathit{var}_{Cong} : v_1 =_\ni v_2 \to \mathit{var}\ v_1 =_\vdash \mathit{var}\ v_2 \\
&\quad \lambda_{Cong} \quad : t_1 =_\vdash t_2 \to \lambda\ t_1 =_\vdash \lambda\ t_2 \\
&\quad (@_{Cong}) : t_{11} =_\vdash t_{12} \to t_{21} =_\vdash t_{22} \to t_{11} @ t_{21} =_\vdash t_{12} @ t_{22} \\
&\quad (/\!\!\!\!\vdash_{Cong}) : t_1 =_\vdash t_2 \to \rho_1 =_\Rightarrow \rho_2 \to t_1 /\!\!\!\!\vdash \rho_1 =_\vdash t_2 /\!\!\!\!\vdash \rho_2 \\
&\quad \text{-- Cast, } \beta \text{ and } \eta \text{ equality.} \\
&\quad \mathit{castEq}_\vdash : t ::^{\equiv}_{\vdash} \mathit{eq} =_\vdash t \\
&\quad \beta \qquad : (\lambda\ t_1) @ t_2 =_\vdash t_1 /\!\!\!\!\vdash \mathit{sub}\ t_2 \\
&\quad \eta \qquad : \{ t : \Gamma \vdash \Pi\ \tau_1\ \tau_2 \} \to \lambda\ ((t /\!\!\!\!\vdash \mathit{wk}\ \tau_1) @ \mathit{var}\ \mathit{vz}) =_\vdash t \\
&\quad \text{-- Substitution application axioms.} \\
&\quad \ldots
\end{aligned}
$$

The η law basically states that, if x is not free in t, and t is of function type, then $\lambda x.t\ x = t$. The first precondition on t is handled by explicitly weakening t, though.

The behaviour of $(/\!\!\!\!\vdash)$ also needs to be postulated. The abstraction and application cases are structural; the *id* case returns the term unchanged, and the $(\odot)$ case is handled by applying the two substitutions one after the other; a variable is weakened by applying *vs* to it; substituting t for variable zero results in t, and otherwise the variable's index is lowered by one; and finally lifted substitutions need to be handled appropriately:

$$
\begin{aligned}
&\textbf{data}\ (=_\vdash) : \Gamma_1 \vdash \tau_1 \to \Gamma_2 \vdash \tau_2 \to \mathit{Set}\ \textbf{where} \\
&\quad \ldots \\
&\quad \text{-- Substitution application axioms.} \\
&\quad \mathit{substLam} \quad : \lambda\ t \quad /\!\!\!\!\vdash \rho \quad =_\vdash \lambda\ (t /\!\!\!\!\vdash \rho \uparrow \tau_1) \\
&\quad \mathit{substApp} \quad : (t_1 @ t_2) \quad /\!\!\!\!\vdash \rho \quad =_\vdash (t_1 /\!\!\!\!\vdash \rho) @ (t_2 /\!\!\!\!\vdash \rho) \\
&\quad \mathit{idVanishesTm} : t \quad /\!\!\!\!\vdash \mathit{id} \quad =_\vdash t \\
&\quad \mathit{compSplitsTm} : t \quad /\!\!\!\!\vdash (\rho_1 \odot \rho_2) \quad =_\vdash t /\!\!\!\!\vdash \rho_1 /\!\!\!\!\vdash \rho_2 \\
&\quad \mathit{substWk} \quad : \mathit{var}\ v \quad /\!\!\!\!\vdash \mathit{wk}\ \sigma \quad =_\vdash \mathit{var}\ (\mathit{vs}\ v) \\
&\quad \mathit{substVzSub} \quad : \mathit{var}\ \mathit{vz} \quad /\!\!\!\!\vdash \mathit{sub}\ t \quad =_\vdash t \\
&\quad \mathit{substVsSub} \quad : \mathit{var}\ (\mathit{vs}\ v) \quad /\!\!\!\!\vdash \mathit{sub}\ t \quad =_\vdash \mathit{var}\ v \\
&\quad \mathit{substVzLift} \quad : \mathit{var}\ \mathit{vz} \quad /\!\!\!\!\vdash (\rho \uparrow \sigma) \quad =_\vdash \mathit{var}\ \mathit{vz} \\
&\quad \mathit{substVsLift} \quad : \mathit{var}\ (\mathit{vs}\ v) \quad /\!\!\!\!\vdash (\rho \uparrow \sigma) \quad =_\vdash \mathit{var}\ v /\!\!\!\!\vdash \rho /\!\!\!\!\vdash \mathit{wk}\ (\sigma\ /\ \rho)
\end{aligned}
$$

3.7 Design Choices

Initially I tried to formalise a language with implicit substitutions, i.e. I tried to implement $(/\!\vdash)$ as a function instead of as a constructor. This turned out to be difficult, since when $(/\!\vdash)$ is defined as a function many substitution lemmas need to be proved in the initial mutually recursive definition containing all the type families above, and when the mutual dependencies become too complicated it is hard to prove that the code is terminating.

As an example of why substitution lemmas are needed, take the axiom *substApp* above. If *substApp* is considered as a pattern matching equation in the definition of $(/\!\vdash)$, then it needs to be modified in order to type check:

$$(t_1 @ t_2) \mathbin{/\!\vdash} \rho \;=\; (t_1 \mathbin{/\!\vdash} \rho) @ (t_2 \mathbin{/\!\vdash} \rho) \mathbin{::^{\equiv}_{\vdash}} \mathit{subCommutes}_\star$$

Here $\mathit{subCommutes}_\star$ states that, in certain situations, *sub* commutes with other substitutions:

$$\mathit{subCommutes}_\star : \tau \;/\; (\rho \uparrow \sigma) \;/\; \mathit{sub}\ (t \mathbin{/\!\vdash} \rho) =_\star \tau \;/\; \mathit{sub}\ t \;/\; \rho$$

Avoidance of substitution lemmas is also the reason for making the equalities heterogeneous. It would be possible to enforce directly that, for instance, two terms are only equal if their respective types are equal. It suffices to add the type equality as an index to the term equality:

$$(=_\vdash) : \{\tau_1 =_\star \tau_2\} \;\to\; \Gamma_1 \vdash \tau_1 \;\to\; \Gamma_2 \vdash \tau_2 \;\to\; \mathit{Set}$$

However, in this case *substApp* could not be defined without a lemma like $\mathit{subCommutes}_\star$. Furthermore this definition of $(=_\vdash)$ easily leads to a situation where two equality *proofs* need to be proved equal. These problems are avoided by, instead of enforcing equality directly, proving that term equality implies type equality ($eq_\vdash eq_\star$) and so on. These results also require lemmas like $\mathit{subCommutes}_\star$, but the lemmas can be proved after the first, mutually recursive definition.

The problems described above could be avoided in another way, by postulating the substitution lemmas needed, i.e. adding them as type equality constructors. This approach has not been pursued, as I have tried to minimise the amount of "unnecessary" postulates and definitions.

The postulate *substApp* discussed above also provides motivation for defining $(/)$ as a function, even though $(/\!\vdash)$ is a constructor: if $(/)$ were a constructor then $t_1 \mathbin{/\!\vdash} \rho$ would not have a Π type as required by $(@)$ (the type would be $\Pi\ \tau_1\ \tau_2\ /\ \rho$), and hence a cast would be required in the definition of *substApp*. I have not examined this approach in detail, but I suspect that it would be harder to work with.

Another important design choice is the basic set of substitutions. The following definition is a natural candidate for this set:

$$\begin{array}{l}
\textbf{data}\ (\tilde{\Rightarrow}) : \mathit{Ctxt} \;\to\; \mathit{Ctxt} \;\to\; \mathit{Set}\ \textbf{where} \\
\quad \emptyset \quad\; : \varepsilon \tilde{\Rightarrow} \Delta \\
\quad (\blacktriangleright) : (\rho : \Gamma \tilde{\Rightarrow} \Delta) \;\to\; \Delta \vdash \tau \;/\; \rho \;\to\; \Gamma \triangleright \tau \tilde{\Rightarrow} \Delta
\end{array}$$

This type family encodes simultaneous (parallel) substitutions; for every variable in the original context a term in the resulting context is given. So far so good, but the substitutions used in the type signatures above (*sub* and *wk*, for instance) need to be implemented in terms of $\emptyset$ and $(\blacktriangleright)$, and these implementations seem to require various substitution lemmas, again leading to the problems described above.

Note that, even though $(\Rightarrow)$ is not used to define what a substitution is, the substitutions $\emptyset$ and $(\blacktriangleright)$ can be defined in terms of the other substitutions, and they are used in Sect. 5.3 when value environments are defined.

4 Removing Explicit Substitutions

In Sect. 5 a normalisation proof for the lambda calculus introduced in Sect. 3 is presented. The normalisation function defined there requires terms without explicit substitutions ("implicit terms"). This section defines a data type Tm^- representing such terms.

The type Tm^- provides a view of the $(\vdash)$ terms (the "explicit terms"). Other views will be introduced later, for instance normal forms (Sect. 5.1), and they will all follow the general scheme employed by Tm^-, with minor variations.

Implicit terms are indexed on explicit terms to which they are, in a sense, $\beta\eta$-equal; the function tm^-ToTm converts an implicit term to the corresponding explicit term, and $tm^-ToTm\ t^- =_\vdash t$ for every implicit term $t^- : Tm^-\ t$:

$$
\begin{array}{l}
\textbf{data}\ Tm^- : \Gamma \vdash \tau \to \mathit{Set}\ \textbf{where} \\
\quad \begin{array}{ll}
var^- & : (v : \Gamma \ni \tau) \to Tm^-\ (var\ v) \\
\lambda^- & : Tm^-\ t \to Tm^-\ (\lambda\ t) \\
(@^-) & : Tm^-\ t_1 \to Tm^-\ t_2 \to Tm^-\ (t_1 @ t_2) \\
(::^{\equiv}_{\vdash^-}) & : Tm^-\ t_1 \to t_1 =_\vdash t_2 \to Tm^-\ t_2
\end{array} \\
\\
\begin{array}{l}
tm^-ToTm : \{\, t : \Gamma \vdash \tau \,\} \to Tm^-\ t \to \Gamma \vdash \tau \\
tm^-ToTm\ (var^-\ v) = var\ v \\
tm^-ToTm\ (\lambda^-\ t^-) = \lambda\ (tm^-ToTm\ t^-) \\
tm^-ToTm\ (t_1^-\ @^-\ t_2^-) = (tm^-ToTm\ t_1^-) @ (tm^-ToTm\ t_2^-) ::^{\equiv}_{\vdash} \ldots \\
tm^-ToTm\ (t^-\ ::^{\equiv}_{\vdash^-}\ eq) = tm^-ToTm\ t^-\ ::^{\equiv}_{\vdash}\ eq_\vdash eq_\star\ eq
\end{array}
\end{array}
$$

(The ellipsis stands for uninteresting code that has been omitted.)

It would be possible to index implicit terms on types instead. However, by indexing on explicit terms soundness results are easily expressed in the types of functions constructing implicit terms. For instance, the function $tmToTm^-$ which converts explicit terms to implicit terms has the type $(t{:}\Gamma \vdash \tau) \to Tm^-\ t$, which guarantees that the result is $\beta\eta$-equal to t. The key to making this work is the cast constructor $(::^{\equiv}_{\vdash^-})$, which makes it possible to include equality proofs in an implicit term; without $(::^{\equiv}_{\vdash^-})$ no implicit term could be indexed on $t \not\vdash \rho$, for instance.

Explicit terms are converted to implicit terms using techniques similar to those in [McBb]. Due to lack of space this conversion is not discussed further here.

5 Normalisation Proof

This section proves that every explicit term has a normal form. The proof uses normalisation by evaluation (NBE). Type-based NBE proceeds as follows:

- First terms (in this case implicit terms) are evaluated by a function $[\![\cdot]\!]$ (Sect. 5.4), resulting in "values" (Sect. 5.2). Termination issues are avoided by representing function types using the function space of the meta-language.
- Then these values are converted to normal forms by using two functions, often called *reify* and *reflect*, defined by recursion on the (spines of the) types of their arguments (Sect. 5.5).

5.1 Normal Forms

Let us begin by defining what a normal form is. Normal forms (actually long $\beta\eta$-normal forms) and atomic forms are defined simultaneously. Both type families are indexed on a $\beta\eta$-equivalent term, just like Tm^- (see Sect. 4):

$$
\begin{array}{l}
\textbf{data}\ \mathit{Atom} : \Gamma \vdash \tau \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{var}_{At} : (v : \Gamma \ni \tau) \to \mathit{Atom}\ (\mathit{var}\ v) \\
\quad (@_{At}) : \mathit{Atom}\ t_1 \to \mathit{NF}\ t_2 \to \mathit{Atom}\ (t_1 @ t_2) \\
\quad (::^{\equiv}_{At}) : \mathit{Atom}\ t_1 \to t_1 =_{\vdash} t_2 \to \mathit{Atom}\ t_2 \\
\textbf{data}\ \mathit{NF} : \Gamma \vdash \tau \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{atom}^{\star}_{NF} : \{\, t : \Gamma \vdash \star \,\} \to \mathit{Atom}\ t \to \mathit{NF}\ t \\
\quad \mathit{atom}^{El}_{NF} : \{\, t : \Gamma \vdash \mathit{El}\ t' \,\} \to \mathit{Atom}\ t \to \mathit{NF}\ t \\
\quad \lambda_{NF} : \mathit{NF}\ t \to \mathit{NF}\ (\lambda\ t) \\
\quad (::^{\equiv}_{NF}) : \mathit{NF}\ t_1 \to t_1 =_{\vdash} t_2 \to \mathit{NF}\ t_2
\end{array}
$$

The two atom_{NF} constructors ensure that the only normal forms of type $\Pi\ \tau_1\ \tau_2$ are lambdas and casts; this is how long η-normality is ensured.

A consequence of the inclusion of the cast constructors $(::^{\equiv}_{At})$ and $(::^{\equiv}_{NF})$ is that normal forms are not unique. However, the equality on normal and atomic forms (congruence plus postulates stating that casts can be removed freely) ensures that equality can be decided by erasing all casts and annotations and then checking syntactic equality.

A normal form can be converted to a term in the obvious way, and the resulting term is $\beta\eta$-equal to the index (cf. $tm^- ToTm$ in Sect. 4):

$$
\begin{array}{l}
\mathit{nfToTm} : \{\, t : \Gamma \vdash \tau \,\} \to \mathit{NF}\ t \to \Gamma \vdash \tau \\
\mathit{nfToTmEq} : (\mathit{nf} : \mathit{NF}\ t) \to \mathit{nfToTm}\ \mathit{nf} =_{\vdash} t
\end{array}
$$

Similar functions are defined for atomic forms.

We also need to weaken normal and atomic forms. In fact, multiple weakenings will be performed at once. In order to express these multi-weakenings context extensions are introduced. The type $\mathit{Ctxt}^+\ \Gamma$ stands for context extensions which can be put "to the right of" the context Γ by using $(+\!\!+)$:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Ctxt}^+\ (\Gamma : \mathit{Ctxt}) : \mathit{Set}\ \textbf{where} \\
\quad \varepsilon^+ \quad\ : \mathit{Ctxt}^+\ \Gamma \\
\quad (\triangleright^+) : (\Gamma' : \mathit{Ctxt}^+\ \Gamma) \to \mathit{Ty}\ (\Gamma +\!\!+ \Gamma') \to \mathit{Ctxt}^+\ \Gamma \\
(+\!\!+) : (\Gamma : \mathit{Ctxt}) \to \mathit{Ctxt}^+\ \Gamma \to \mathit{Ctxt} \\
\Gamma +\!\!+ \varepsilon^+ \qquad\ = \Gamma \\
\Gamma +\!\!+ (\Gamma' \triangleright^+ \tau) = (\Gamma +\!\!+ \Gamma') \triangleright \tau
\end{array}
$$

Now the following type signatures can be understood:

$$
\begin{array}{l}
\mathit{wk}^* \quad : (\Gamma' : \mathit{Ctxt}^+\ \Gamma) \to \Gamma \Rightarrow \Gamma +\!\!+ \Gamma' \\
\mathit{wk}^*_{At} : \mathit{Atom}\ t \to (\Gamma' : \mathit{Ctxt}^+\ \Gamma) \to \mathit{Atom}\ (t\ /\!\vdash\ \mathit{wk}^*\ \Gamma')
\end{array}
$$

5.2 Values

Values are represented using one constructor for each type constructor, plus a case for casts (along the lines of previously introduced types indexed on terms). Values of function type are represented using meta-language functions:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Val} : \Gamma \vdash \tau \to \mathit{Set}\ \textbf{where} \\
\quad (::_{\mathit{Val}}) : \mathit{Val}\ t_1 \to t_1 =_\vdash t_2 \to \mathit{Val}\ t_2 \\
\quad \star_{\mathit{Val}} \ : \{\, t : \Gamma \vdash \star \,\} \qquad \to \mathit{Atom}\ t \to \mathit{Val}\ t \\
\quad \mathit{El}_{\mathit{Val}} \ : \{\, t : \Gamma \vdash \mathit{El}\ t' \,\} \to \mathit{Atom}\ t \to \mathit{Val}\ t \\
\quad \Pi_{\mathit{Val}} \ : \quad \{\, t_1 : \Gamma \vdash \Pi\ \tau_1\ \tau_2 \,\} \\
\qquad\quad \to (f \ : \ (\Gamma' : \mathit{Ctxt}^+\ \Gamma) \\
\qquad\qquad\quad \to \{\, t_2 : \Gamma +\!\!+ \Gamma' \vdash \tau_1\ /\ \mathit{wk}^*\ \Gamma' \,\} \\
\qquad\qquad\quad \to (v : \mathit{Val}\ t_2) \\
\qquad\qquad\quad \to \mathit{Val}\ ((t_1\ /\!\vdash\ \mathit{wk}^*\ \Gamma')@t_2)) \\
\qquad\quad \to \mathit{Val}\ t_1
\end{array}
$$

The function f given to $\Pi_{\mathit{Val}}\,\{\,t_1\,\}$ essentially takes an argument value and evaluates t_1 applied to this argument. For technical reasons, however, we need to be able to weaken t_1 (see *reify* in Sect. 5.5). This makes *Val* look suspiciously like a Kripke model [MM91] (suitably generalised to a dependently typed setting); this has not been verified in detail, though. The application operation of this supposed model is defined as follows. Notice that the function component of Π_{Val} is applied to an empty Ctxt^+ here:

$$
\begin{array}{l}
(@_{\mathit{Val}}) : \mathit{Val}\ t_1 \to \mathit{Val}\ t_2 \to \mathit{Val}\ (t_1@t_2) \\
\Pi_{\mathit{Val}}\ f \qquad\quad @_{\mathit{Val}}\ v_2 = f\ \varepsilon^+\ (v_2\ ::_{\mathit{Val}}\ \ldots)\ ::_{\mathit{Val}}\ \ldots \\
(v_1\ ::_{\mathit{Val}}\ \mathit{eq})\ @_{\mathit{Val}}\ v_2 = (v_1\ @_{\mathit{Val}}\ (v_2\ ::_{\mathit{Val}}\ \ldots))\ ::_{\mathit{Val}}\ \ldots
\end{array}
$$

The transition function of the model weakens values:

$$
\mathit{wk}^\star_{\mathit{Val}} : \mathit{Val}\ t \to (\Gamma' : \mathit{Ctxt}^+\ \Gamma) \to \mathit{Val}\ (t\ /\!\vdash\ \mathit{wk}^*\ \Gamma')
$$

Note that *Val* is not a positive data type, due to the negative occurrence of *Val* inside of Π_{Val}, so this data type is not part of the treatment in [DS06]. In

practise this should not be problematic, since the type index of that occurrence, $\tau_1 \mathbin{/} wk^* \ \Gamma'$, is smaller than the type index of $\Pi_{Val} \ f$, which is $\Pi \ \tau_1 \ \tau_2$. Here we count just the *spine* of the type, ignoring the contents of *El*, so that τ and $\tau \mathbin{/} \rho$ have the same size, and two equal types also have the same size. In fact, by supplying a spine argument explicitly it should not be difficult to define *Val* as a structurally recursive function instead of as a data type.

5.3 Environments

The function $[\![\cdot]\!]$, defined in Sect. 5.4, makes use of environments, which are basically substitutions containing values instead of terms:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Env} : \Gamma \Rightarrow \Delta \ \to\ \mathit{Set}\ \textbf{where} \\
\quad \emptyset_{Env} \quad\ : \mathit{Env}\ \emptyset \\
\quad (\blacktriangleright_{Env}) : \mathit{Env}\ \rho \ \to\ \mathit{Val}\ t \ \to\ \mathit{Env}\ (\rho \blacktriangleright t) \\
\quad (::^{\equiv}_{Env}) : \mathit{Env}\ \rho_1 \ \to\ \rho_1 =_{\Rightarrow} \rho_2 \ \to\ \mathit{Env}\ \rho_2
\end{array}
$$

Note that the substitutions $\emptyset$ and $(\blacktriangleright)$ from Sect. 3.7 are used as indices here.

It is straight-forward to define functions for looking up a variable in an environment and weakening an environment:

$$
\begin{array}{l}
\mathit{lookup} : (v : \Gamma \ni \tau) \ \to\ \mathit{Env}\ \rho \ \to\ \mathit{Val}\ (\mathit{var}\ v \mathbin{/\!\!\vdash} \rho) \\
wk^{\star}_{Env} : \mathit{Env}\ \rho \ \to\ (\Delta' : \mathit{Ctxt}^{+}\ \Delta) \ \to\ \mathit{Env}\ (\rho \odot wk^*\ \Delta')
\end{array}
$$

5.4 Evaluating Terms

Now we can evaluate an implicit term, i.e. convert it to a value. The most interesting case is $\lambda^-\ t_1^-$, where t_1^- is evaluated in an extended, weakened environment:

$$
\begin{array}{l}
[\![\cdot]\!] : \mathit{Tm}^-\ t \ \to\ \mathit{Env}\ \rho \ \to\ \mathit{Val}\ (t \mathbin{/\!\!\vdash} \rho) \\
[\![\mathit{var}^-\ v]\!]\,\gamma \quad\ = \mathit{lookup}\ v\ \gamma \\
[\![t_1^-\ @^-\ t_2^-]\!]\,\gamma \ = ([\![t_1^-]\!]\,\gamma \ @_{Val}\ [\![t_2^-]\!]\,\gamma) \ ::_{Val}\ \ldots \\
[\![t^- ::^{\equiv}_{\vdash -}\ \mathit{eq}]\!]\,\gamma = [\![t^-]\!]\,\gamma \ ::_{Val}\ \ldots \\
[\![\lambda^-\ t_1^-]\!]\,\gamma \quad\ = \\
\quad \Pi_{Val}\,(\backslash \Delta'\ v_2 \ \to\ [\![t_1^-]\!]\,(wk^{\star}_{Env}\ \gamma\ \Delta' \blacktriangleright_{Env} (v_2 \ ::_{Val} \ldots)) \ ::_{Val}\ \ldots \beta \ldots)
\end{array}
$$

(The notation $\backslash x \to \ldots$ is lambda abstraction in the meta-language.) It would probably be straightforward to evaluate explicit terms directly, without going through implicit terms (cf. [Coq02]). Here I have chosen to separate these two steps, though.

5.5 *reify* and *reflect*

Let us now define *reify* and *reflect*. These functions are implemented by recursion over spines (see Sect. 5.2), in order to make them structurally recursive, but to avoid clutter the spine arguments are not written out below.

The interesting cases correspond to function types, for which *reify* and *reflect* use each other recursively. Notice how *reify* applies the function component of Π_{Val} to a singleton $Ctxt^{+}$, to enable using the reflection of variable zero, which has a weakened type; this is the reason for including weakening in the definition of Π_{Val}:

$$
\begin{aligned}
&\mathit{reify} : (\tau : \mathit{Ty}\ \Gamma) \to \{\, t : \Gamma \vdash \tau \,\} \to \mathit{Val}\ t \to \mathit{NF}\ t \\
&\mathit{reify}\ (\Pi\ \tau_1\ \tau_2)\ (\Pi_{Val}\ f) = \\
&\quad \lambda_{NF}\ (\mathit{reify}\ (\tau_2\ /\ \_\ /\ \_) \\
&\qquad\qquad (f\ (\varepsilon^{+} \triangleright^{+} \tau_1)\ (\mathit{reflect}\ (\tau_1\ /\ \_)\ (\mathit{var}_{At}\ \mathit{vz})\ ::_{Val}\ \ldots))) \\
&\quad ::_{NF}\ \ldots \eta \ldots \\
&\mathit{reflect} : (\tau : \mathit{Ty}\ \Gamma) \to \{\, t : \Gamma \vdash \tau \,\} \to \mathit{Atom}\ t \to \mathit{Val}\ t \\
&\mathit{reflect}\ (\Pi\ \tau_1\ \tau_2)\ \mathit{at} = \Pi_{Val}\ (\backslash \Gamma'\ v \to \\
&\quad \mathit{reflect}\ (\tau_2\ /\ \_\ /\ \_)\ (\mathit{wk}^{*}_{At}\ \mathit{at}\ \Gamma'\ @_{At}\ \mathit{reify}\ (\tau_1\ /\ \_)\ v))
\end{aligned}
$$

Above underscores (_) have been used instead of giving non-hidden arguments which can be inferred automatically by the AgdaLight type checker, and some simple and boring cases have been omitted to save space.

5.6 Normalisation

After having defined $[\![\cdot]\!]$ and *reify* it is very easy to normalise a term. First we build an identity environment by applying *reflect* to all the variables in the context:

$$\mathit{id}_{Env} : (\Gamma : \mathit{Ctxt}) \to \mathit{Env}\ (\mathit{id}\ \Gamma)$$

Then an explicit term can be normalised by converting it to an implicit term, evaluating the result in the identity environment, and then reifying:

$$
\begin{aligned}
&\mathit{normalise} : (t : \Gamma \vdash \tau) \to \mathit{NF}\ t \\
&\mathit{normalise}\ t = \mathit{reify}\ \_\ ([\![\mathit{tmToTm}^{-}\ t]\!]\ (\mathit{id}_{Env}\ \_)\ ::_{Val}\ \ldots)
\end{aligned}
$$

Since a normal form is indexed on an equivalent term it is easy to show that *normalise* is sound:

$$
\begin{aligned}
&\mathit{normaliseEq} : (t : \Gamma \vdash \tau) \to \mathit{nfToTm}\ (\mathit{normalise}\ t) =_{\vdash} t \\
&\mathit{normaliseEq}\ t = \mathit{nfToTmEq}\ (\mathit{normalise}\ t)
\end{aligned}
$$

If this normalising function is to be really useful (as part of a type checker, for instance) it should also be proved, for the normal form equality ($=_{NF}$), that $t_1 =_{\vdash} t_2$ implies that $\mathit{normalise}\ t_1 =_{NF} \mathit{normalise}\ t_2$. This is left for future work, though.

6 Related Work

As stated in the introduction Dybjer's formalisation of categories with families [Dyb96] seems to be the only prior example of a formalisation of a dependently

typed language done using the internal approach to handle the type system invariants. Other formalisations of dependently typed languages, such as McKinna/Pollack [MP99] and Barras/Werner [BW97], have used the external approach. There is also an example, due to Adams [Ada04], of a hybrid approach which handles some invariants internally, but not the type system.

Normalisation by evaluation (NBE) seems to have been discovered independently by Martin-Löf (for a version of his type theory) [ML75] and Berger and Schwichtenberg (for simply typed lambda calculus) [BS91]. Martin-Löf has also defined an NBE algorithm for his logical framework [ML04], and recently Dybjer, Abel and Aehlig have done the same for Martin-Löf type theory with one universe [AAD07].

NBE has been formalised, using the internal approach, by T. Coquand and Dybjer, who treated a combinatory version of Gödel's System T [CD97]. C. Coquand has formalised normalisation for a simply typed lambda calculus with explicit substitutions, also using the internal approach [Coq02]. Her normalisation proof uses NBE and Kripke models, and in that respect it bears much resemblance to this one. McBride has implemented NBE for the untyped lambda calculus [McBa]. His implementation uses an internal approach (nested types in Haskell) to ensure that terms are well-scoped, and that aspect of his code is similar to mine.

My work seems to be the first formalised NBE algorithm for a dependently typed language.

7 Discussion

I have presented a formalisation of a dependently typed lambda calculus, including a proof of normalisation, using the internal approach to handle typing rules. This formalisation demonstrates that, at least in this case, it is feasible to use the internal approach when programming with invariants strong enough to encode the typing rules of a dependently typed language. How this method compares to other approaches is a more difficult question, which I do not attempt to answer here.

Acknowledgements

I am grateful to Ulf Norell, who has taught me a lot about dependently typed programming, discussed many aspects of this work, and fixed many of the bugs in AgdaLight that I have reported. I would also like to thank Thierry Coquand and Peter Dybjer for interesting discussions and useful pointers, and Patrik Jansson and the anonymous referees for helping me with the presentation.

References

[AA02] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.

[AAD07] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. Submitted for publication, 2007.

[AC06] Thorsten Altenkirch and James Chapman. Tait in one big step. In *MSFP 2006*, 2006.

[Ada04] Robin Adams. Formalized metatheory with terms represented by an indexed family of types. In *TYPES 2004*, volume 3839 of *LNCS*, pages 1–16, 2004.

[AR99] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL '99*, volume 1683 of *LNCS*, pages 453–468, 1999.

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *LICS '91*, pages 203–211, 1991.

[BW97] Bruno Barras and Benjamin Werner. Coq in Coq. Unpublished, 1997.

[CD97] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7(1):75–94, 1997.

[Coq02] Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15:57–90, 2002.

[Dan07] Nils Anders Danielsson. Personal web page. Available at `http://www.cs.chalmers.se/~nad/`, 2007.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.

[Dyb96] Peter Dybjer. Internal type theory. In *TYPES '95*, volume 1158 of *LNCS*, pages 120–134, 1996.

[McBa] Conor McBride. Beta-normalization for untyped lambda-calculus. Unpublished program.

[McBb] Conor McBride. Type-preserving renaming and substitution. Unpublished.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[ML04] Per Martin-Löf. Normalization by evaluation and by the method of computability. Lecture series given at Logikseminariet Stockholm–Uppsala, 2004.

[MM91] John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51:99–124, 1991.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[MP99] James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23(3):373–409, 1999.

[MW06] James McKinna and Joel Wright. A type-correct, stack-safe, provably correct expression compiler in Epigram. Accepted for publication in the Journal of Functional Programming, 2006.

[Nor07] Ulf Norell. AgdaLight home page. Available at `http://www.cs.chalmers.se/~ulfn/agdaLight/`, 2007.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory, An Introduction*. Oxford University Press, 1990.

[PL04] Emir Pašalić and Nathan Linger. Meta-programming with typed object-language representations. In *GPCE 2004*, volume 3286 of *LNCS*, pages 136–167, 2004.

[XCC03] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *POPL '03*, pages 224–235, 2003.