A formalization of Γ_∞ in Coq

Robbert Krebbers [*]

March 16, 2010

Abstract

In this paper we present a formalization of the type systems Γ_∞ in the proof assistant Coq. The family of type systems Γ_∞, described in a recent article by Geuvers, McKinna and Wiedijk [9], presents type theory without the need for explicit contexts. A typing judgment in Γ_∞ is of the shape $A :_\infty B$ while an ordinary judgment is of the shape $\Gamma \vdash A : B$.

This approach of Geuvers *et al.* makes a bridge between traditional logic and type theory. In the former free variables are really free and contexts are non-explicit, as in Γ_∞. Furthermore Γ_∞ could make it possible to create a stateless version of an LCF style prover.

The important part of [9] is a theorem that states that there is a natural correspondence between judgments in Γ_∞ and ordinary Pure Type Systems. Their paper contains an informal proof of this theorem. We have formalized many of their definitions and lemmas which result in a proof of one direction of this correspondence theorem.

1 Introduction

A judgment in type theory is traditionally of the shape $\Gamma \vdash A : B$ where Γ is a context which gives the types of free variables occurring in the terms A and B. In a recent article [9], Geuvers, McKinna and Wiedijk have described an approach to present type theory without the use of explicit contexts. A typing judgment in Γ_∞ is of the shape $A :_\infty B$. One should think of these judgments as $\Gamma_\infty \vdash A : B$ where Γ_∞ is a fixed infinite context which has infinitely many variables for each possible type. Geuvers *et al.* have informally proven that these systems have exactly the same type correct terms as ordinary Pure Type Systems.

In this research we have formalized the infrastructure of Γ_∞ and proven one direction of this correspondence theorem in Coq. Our development is inspired by the framework of Aydemir *et al.* for formalization of metatheory in Coq [2]. In this paper we present our formalization, discuss the issues we have occurred and how we have dealt with those issues. Furthermore we will answer the following questions:

1. What insights does formalizing Γ_∞ give to formalizing metatheory in general?

[*] Student number: s0513229, e-mail: robbertkrebbers@student.ru.nl, this research is supervised by James McKinna (james@cs.ru.nl).

2. How suitable is the methodology of Aydemir *et al.*?

1.1 Outline

This paper is outlined as follows. The rest of this section motivates why Γ_∞ and a formalization of it are useful. Furthermore we survey previous work on this topic and describe the typography used in this paper. Section 2 until 5 describe the formalized infrastructure, some formalization issues and important theorems. In Section 6 we describe the correspondence theorem. In Section 7 we conclude with answers to the questions stated in the previous paragraph and give directions for further research. For the proofs of all theorems in this paper we refer to our formal developments.

1.2 Motivation

Pure Type Systems (PTSs) are a generalization of many existing and commonly used systems like $\lambda{\rightarrow}$, λP, system F, system F_ω and the Calculus of Constructions (the basis of the Calculus of Inductive Constructions, the type theory of Coq). Due to the Curry-Howard-de Bruijn correspondence Pure Type Systems correspond to intuitionistic logics.

The main difference between type theory and traditional logic is that in type theory *free variables* are bound to an explicit context, while in logic free variables are really 'free': they are taken from an infinite collection of 'free' variables. The systems of Geuvers *et al.* attempt to made a bridge between type theory and logic by making type theory look more like traditional logic [9]. Another interesting observation is that Γ_∞ makes a difference between free and bound variables on the level of pseudo-terms already, this approach turns out to be very convenient for a formalization.

Another reason why Γ_∞ is interesting can be found in the application of its theory in the architecture of a proof assistant. Many proof assistants, for example Coq [8] and Hol Light [10], are based on an LCF architecture. This means that the type checking kernel contains a small number of functions that can be used to construct well typed terms. Ideally such kernel contains a function app that constructs the application of two (well typed) terms and throws an exception if this is not possible [9].

```
app : term * term -> term
```

Typically a function like app needs to check whether the contexts are compatible. However in realistic systems this is a very expensive job because contexts are very big since they contain all the previously processed definitions [9]. Therefore the kernel contains an abstract type, the environment, which represents the context. Although it is perfectly possible to build a kernel in a purely functional fashion (as the Coq kernel), the environment is usually stored in a global variable that corresponds to the 'state' of the system [9]. This 'state' makes reasoning about the system harder, Γ_∞ could make it possible to realize a completely 'stateless' architecture.

In the previous paragraphs we have summarized the possible uses of Γ_∞, now one may wonder why it is useful to formalize this system. After all a formalization does not give full confidence in the system's correctness, because absolute correctness cannot be obtained by any means at all [16]. However a formalization gives more confidence than an informal proof. A formalization is especially valuable for a system that is fairly new and thus has not been discussed in the literature yet. Besides [9] left many details of definitions and proofs implicit, a formalization fills in these gaps.

Because the author has solely experience with the proof assistants Coq [8] and Mizar [14] we have used Coq for this development. Mizar is less suitable for our developments because it does not contain the machinery for inductive types. Despite the choice for Coq, our development does not require much of its specific infrastructure. Our development only relies on libraries for finite sets and lists therefore this research could be repeated in any proof assistant with support for inductive types.

1.3 Previous work

Other research such as [13, 1] has already shown that it is possible to formalize big parts of the theory of Pure Type Systems in a proof assistant. Unfortunately none of these formalizations are suitable to adapt for our developments. McKinna and Pollack's formalization [13] is written in Lego [15], a proof assistant whose development has stopped over 10 years ago. Adams' formalization [1] is written in Coq but does not distinct between free and bound variables and uses solely De Bruijn indexes what makes it unsuitable for Γ_∞.

Bruno Barras' formalization of Coq in Coq [4] also uses solely De Bruijn indexes and is written for an ancient version of Coq. Therefore we have chosen to use the framework of Aydemir *et al.* in Coq [2] as a starting point for our developments. This framework contains formalizations of $\lambda{\rightarrow}$, the Calculus of Constructions and parts of ML.

1.4 Typography

This paper uses informal notations resulting from a manual translation from our Coq sources into LaTeX. Names in `verbatim` style correspond to names of definitions and lemmas in our formal development. Furthermore we use the usual logical symbols like $\wedge$, $\implies$, $\forall$ and set operations like $\emptyset$, $\cup$ and $\in$. Moreover we drop type labels (e.g. for the range of logical quantifiers) almost everywhere. Because we are dealing with two languages, the terms of ordinary PTSs and the terms of Γ_∞, we denote constructors explicitly, for example we write (`psrt` s) or (`asrt` s) instead of s to avoid any confusion.

2 Terms

In this section we present our formalization of usual lambda terms and lambda terms in Γ_∞-style (from now on *type annotated terms*). First we discuss various methods how one could deal with bound variables and what method we have chosen to use, then we focus on specific problems we have encountered in our developments.

The standard approach, most commonly used on paper, is to represent all variables as strings. This approach has various problems: one has to deal with variable capture (e.g. in the definition of substitution) and α-equivalence has to be defined explicitly. Moreover the following example shows that this approach is error-prone.

Example 2.1. *Consider the term $\lambda y.(\lambda xy.xy)y$, we should be careful that we do not β-reduce it to nonsense like $\lambda y \lambda y.yy$. We should rename the inner occurrence of y to a fresh name, for example y', this results in the correct β-normal form $\lambda yy'.yy'$.*

McKinna and Pollack [13] presented a method which makes a difference between free and bound variables in pseudo-terms already. This solves the problem of variable capture but α-equivalence still has to be defined explicitly (or be avoided). De Bruijn indexes [7] are one of the most standard methods to solve that problem. Variables are encoded by natural numbers referring to the depth of their binder, this method gives a unique representation of each α-equivalence class.

Example 2.2. *Consider the term $\lambda xy.yx$, this term is represented with De Bruijn indexes*[1] *by $\lambda\lambda 01$.*

As you can see this is very unnatural to read and reasoning about free variables is even worse. The locally nameless solution with de Bruijn indexes for bound variables and names for free variables, discussed in [12, 2], combines the advantages of De Bruijn variables with usual naming of free variables.

Having a difference between bound and free variables is a good thing because that is an essential feature of Γ_∞. However there is a major difference between free variables in Γ_∞ and free variables in the usual lambda calculus, in Γ_∞ free variables x^A are annotated by their types A. Thus we have to deal with a different language of pseudo-terms for both systems. Having separate languages for each system results in a lot of duplicate code, all basic operations (e.g. free variable substitution and opening of a De Bruijn index) and lemmas about those operations need to be defined twice. Hence we discuss some methods to describe these two languages in one language of pseudo-terms.

1. We could use a different constructor for each kind of free variable in the definition of pseudo-terms, this results in pseudo-terms of the following shape.
$$\mathcal{T} ::= s \mid n \mid x \mid x^{\mathcal{T}} \mid \Pi\mathcal{T}.\mathcal{T} \mid \lambda\mathcal{T}.\mathcal{T} \mid \mathcal{T}\mathcal{T}$$
With the closure predicates (Section 2.1) we have to ensure that no free variables of the wrong kind are used. Unfortunately this approach makes it impossible to define a set containing just type annotated variables. Furthermore the extra constructor results in clutter in definitions and proofs.

2. We could use a mutual inductive type of the following shape.
$$\mathcal{T} ::= s \mid n \mid x \mid \mathcal{V}_\infty \mid \Pi\mathcal{T}.\mathcal{T} \mid \lambda\mathcal{T}.\mathcal{T} \mid \mathcal{T}\mathcal{T} \quad \text{with} \quad \mathcal{V}_\infty ::= x^{\mathcal{T}}$$
Although this type basically represents the same language as the previous type the mutual type makes it possible to create sets containing solely type annotated variables, unfortunately we still have that extra constructor.

[1] Note that De Bruijn originally numbers from 1 instead of 0.

3. We could define pseudo terms polymorphically, this results in pseudo terms of the following shape.

$$\mathcal{T}(v) ::= s \mid n \mid v \mid \Pi\mathcal{T}(v).\mathcal{T}(v) \mid \lambda\mathcal{T}(v).\mathcal{T}(v) \mid \mathcal{T}(v)\mathcal{T}(v)$$

   Let $\mathcal{V}_\infty ::= x^{\mathcal{T}(\mathcal{V}_\infty)}$. The types $\mathcal{T}(x)$ and $\mathcal{T}(\mathcal{V}_\infty)$ represent usual pseudo-terms respectively type annotated pseudo-terms.

Definition 2.3. *The* pseudo-terms *(formally* `trm`*) are defined as follows.*

$$\begin{aligned}
\texttt{binder} &::= \texttt{abs} \mid \texttt{pi}\\
\texttt{trm}\ \mathit{var} &::= \texttt{srt} \mid \texttt{bvr nat} \mid \texttt{fvr}\ \mathit{var}\\
&\quad \mid \texttt{bnd binder}\ (\texttt{trm}\ \mathit{var})\ (\texttt{trm}\ \mathit{var})\\
&\quad \mid \texttt{app}\ (\texttt{trm}\ \mathit{var})\ (\texttt{trm}\ \mathit{var})\\
\texttt{avar} &::= \texttt{avr var}\ (\texttt{trm avar})
\end{aligned}$$

Here `srt` *and* `var`[2] *have decidable equality. Finite sets of* `var` *and* `avar` *are named* `vars` *respectively* `avars`*. Furthermore we make the following assumption which expresses that* `var` *has infinitely many elements.*

$$\texttt{var\_fresh} : \forall L \in \texttt{vars}\ .\ \exists x \in \texttt{var}\ .\ x \notin L$$

When doing case analysis on terms we often want to say that λ and Π behave in exactly the same way. The type `binder` allows us to combine these cases and thus saves a lot of copying and pasting [13].

Definition 2.4. Usual pseudo-terms *and* type annotated pseudo-terms *are instances of pseudo-terms.*

$$\texttt{ptrm} ::= \texttt{trm var} \qquad \texttt{atrm} ::= \texttt{trm avar}$$

Despite that the definition `avar` ::= `avr var` (`trm avar`) might looks innocent it is very complicated: it contains "nested" recursive dependencies between pseudo-terms and variables. Unfortunately this resulted in some implementation issues in `Coq` which are addressed now.

1. The induction scheme for `atrm` which `Coq` automatically generates is often not strong enough because `avr` contains sub-terms wherefore we may need an induction hypothesis. Even with the `Scheme` command `Coq` is unable to generate the required induction scheme automatically [6], therefore we had to define it manually.

   Also `Coq`'s `decide equality` tactic is unable to automate the proof of the theorem stating that `atrm` has decidable equality. We had to prove this manually by double induction using the previously defined "nested" induction scheme.

2. We would like to define a polymorphic function that computes the free variables of a term, its signature should be as follows.

$$\texttt{fv} : \forall \mathit{var}\ .\ \texttt{trm}\ \mathit{var} \to \texttt{finset}\ \mathit{var}$$

[2]We abstract from the actual implementation of these types. It is possible to implement these types using the natural numbers as previous research such as [2] has shown.

Unfortunately it is impossible to create polymorphic finite sets with Coq's finite set library, one has to make an instance of the finite set module (which requires a proof of decidable equality) for each kind separately. Type classes may solve this problem, but unfortunately the author was unable to try so because Coq's finite set library is not implemented using type classes (yet).

3. It is impossible to define the hereditary free variables as follows.

$$\begin{aligned} \texttt{hfv}\ (\texttt{asrt}\ s) = \texttt{hfv}\ (\texttt{abvr}\ n) &= \emptyset \\ \texttt{hfv}\ (\texttt{afvr}\ (\texttt{avr}\ v\ t)) &= \{\texttt{avr}\ v\ t\} \cup (\texttt{hfv}\ t) \qquad (\dagger) \\ \texttt{hfv}\ (\texttt{abnd}\ b\ t_1\ t_2) = \texttt{hfv}\ (\texttt{aapp}\ t_1\ t_2) &= (\texttt{hfv}\ t_1) \cup (\texttt{hfv}\ t_2) \end{aligned}$$

In (†) it is obvious that t is a sub-term of `afvr` (`avr` v t), but Coq disagrees. In the general case constructions like this can result in ill-defined functions, hence Coq does not allow it [11]. We have worked around this by defining the function `hfv'` which is parametrized by the function `hfv_avar` [11].

Definition 2.5. *The* free variables `fv` t *of a pseudo-term* t *are defined as follows.*

$$\begin{aligned} \texttt{fv}\ (\texttt{psrt}\ s) = \texttt{fv}\ (\texttt{pbvr}\ s) &= \emptyset \\ \texttt{fv}\ (\texttt{pfvr}\ x) &= \{x\} \\ \texttt{fv}\ (\texttt{pbnd}\ b\ t_1\ t_2) = \texttt{fv}\ (\texttt{papp}\ t_1\ t_2) &= (\texttt{fv}\ t_1) \cup (\texttt{fv}\ t_2) \end{aligned}$$

We need various notions of *free variables* on type annotated terms because we want to obtain free variables with and without type labels. Besides there is no way to easily map between finite sets of different types using Coq's finite set library. Since these notions differ for the constructor `afvr` solely we describe just these cases. Note that `hfv` and `ahfv` are actually defined by a parametrized function.

Definition 2.6. $\texttt{fva}\ (\texttt{afvr}\ (\texttt{avr}\ v\ t)) = \{v\}$

Definition 2.7. $\texttt{afva}\ (\texttt{afvr}\ (\texttt{avr}\ v\ t)) = \{\texttt{avr}\ v\ t\}$

Definition 2.8. $\texttt{hfv}\ (\texttt{afvr}\ (\texttt{avr}\ v\ t)) = \{v\} \cup (\texttt{hfv}\ t)$

Definition 2.9. $\texttt{ahfv}\ (\texttt{afvr}\ (\texttt{avr}\ v\ t)) = \{\texttt{avr}\ v\ t\} \cup (\texttt{ahfv}\ t)$

Furthermore we need the notion of hereditary free type variables [9].

Definition 2.10. $\texttt{ahfv\_type}\ (\texttt{afvr}\ (\texttt{avr}\ v\ t)) = \texttt{ahfv}\ t$

Some obvious correspondence lemmas:

Lemma 2.11. $\texttt{fva\_afva} : x \in \texttt{fva}\ t \iff \exists A.\texttt{avr}\ x\ A \in \texttt{afva}\ t$

Lemma 2.12. $\texttt{hfv\_ahfv} : x \in \texttt{hfv}\ t \iff \exists A.\texttt{avr}\ x\ A \in \texttt{ahfv}\ t$

Lemma 2.13. $\texttt{ahfv\_type\_subseteq\_ahfv} : x \in \texttt{ahfv\_type}\ t \implies x \in \texttt{ahfv}\ t$

We would like to worry about the implementation of bound variables by De Bruijn indexes as little as possible, therefore we define the following operation.

Definition 2.14. Opening *of a De Bruijn index k for u in t (formally* `open_rec`*) is defined as follows.*

$$\begin{aligned}
\{k \to u\}\ \texttt{srt}\ s &= \texttt{srt}\ s \\
\{k \to u\}\ \texttt{bvr}\ i &= \texttt{if}\ (k = i)\ \texttt{then}\ u\ \texttt{else}\ (\texttt{bvr}\ i) \\
\{k \to u\}\ \texttt{fvr}\ x &= \texttt{fvr}\ x \\
\{k \to u\}\ \texttt{bnd}\ b\ t_1\ t_2 &= \texttt{bnd}\ b\ \{k \to u\}t_1\ \{k+1 \to u\}t_2 \\
\{k \to u\}\ \texttt{app}\ t_1\ t_2 &= \texttt{app}\ \{k \to u\}t_1\ \{k \to u\}t_2
\end{aligned}$$

Notation 2.15. *We write t^u (formally* `open`*) for $\{0 \to u\}\ t$.*

Definition 2.16. Substitution *of a free variable x for u in t (formally* `subst`*) is defined as follows.*

$$\begin{aligned}
[x \to u]\ \texttt{srt}\ s &= \texttt{srt}\ s \\
[x \to u]\ \texttt{bvr}\ i &= \texttt{bvr}\ i \\
[x \to u]\ \texttt{fvr}\ y &= \texttt{if}\ (x = y)\ \texttt{then}\ u\ \texttt{else}\ (\texttt{fvr}\ y) \\
[x \to u]\ \texttt{bnd}\ b\ t_1\ t_2 &= \texttt{bnd}\ b\ [x \to u]t_1\ [x \to u]t_2 \\
[x \to u]\ \texttt{app}\ t_1\ t_2 &= \texttt{app}\ [x \to u]t_1\ [x \to u]t_2
\end{aligned}$$

2.1 Local closure

Unfortunately the representation of terms presented in the previous section is not isomorphic to well-formed lambda terms.

Example 2.17. *Consider the term* `bvr` 0*, here 0 does not resolve to any binder and therefore this is not a well-formed lambda term.*

Therefore we define the predicates `pterm` and `aterm` which select the well-formed terms. We use co-finite quantification presented by Aydemir *et al.* [2], this method gives a strong induction principle without having to worry about free variables often.

Definition 2.18. *A pseudo-term $t \in$* `ptrm` *is* locally closed *if* `pterm` t*, where* `pterm` *is defined by the rules in Figure 1.*

$$\frac{}{\texttt{pterm}\ (\texttt{psrt}\ s)} \qquad \frac{}{\texttt{pterm}\ (\texttt{pfvr}\ x)}$$

(a) `term_srt` (b) `term_fvr`

$$\frac{\texttt{pterm}\ t_1 \qquad \forall x \notin L.\texttt{pterm}\ t_2^x}{\texttt{pterm}\ (\texttt{pbnd}\ b\ t_1\ t_2)} \qquad \frac{\texttt{pterm}\ t_1 \qquad \texttt{pterm}\ t_2}{\texttt{pterm}\ (\texttt{papp}\ t_1\ t_2)}$$

(c) `term_bnd` (d) `term_app`

Figure 1: Closure of usual lambda terms.

A more commonly used approach to describe the closed terms is the exists-fresh approach, here the `bnd`-rule is defined as follows.

$$\frac{\texttt{pterm}\ t_1 \qquad \exists x \notin \texttt{fv}\ t_2\ .\ \texttt{pterm}\ t_2^x}{\texttt{pterm}\ (\texttt{pbnd}\ b\ t_1\ t_2)}$$

We have proven that closed terms described by the exists-fresh approach correspond exactly to closed terms described by the co-finite quantification approach. For a more extensive discussion about various approaches to describe closed terms and their correspondence we refer to [2]. Now that we have defined the notion of closed terms we are able to prove various important lemmas.

Lemma 2.19. `psubst_pterm` : $\texttt{pterm}\ t \wedge \texttt{pterm}\ u \implies \texttt{pterm}\ [x \to u]t$

Lemma 2.20. `psubst_intro` : $x \notin \texttt{fv}\ t \wedge \texttt{pterm}\ u \implies t^u = [x \to u]t^x$

Lemma 2.21. `open_pterm` : $\texttt{pterm}\ t^x \wedge \texttt{pterm}\ u \implies \texttt{pterm}\ t^u$

Lemma 2.22. `open_var_unique` : $x \notin \texttt{fv}\ t_1 \wedge x \notin \texttt{fv}\ t_2 \wedge t_1^x = t_2^x \implies t_1 = t_2$

Lemma 2.23. `shape_pterm` : $\texttt{pterm}\ t \implies \exists u. t = u^x$.

Now we define a similar predicate for type annotated terms. Note that such notion of closure is also required for the original representation described in [9], but unfortunately, despite its importance, it is left implicit.

Definition 2.24. *A type annotated pseudo-term $t \in$ `atrm` is* locally closed *if* `aterm` t*, where* `aterm` *is a modification of* `pterm`*. The relevant changes are shown in Figure 2.*

$$\frac{\texttt{aterm}\ t}{\texttt{aterm}\ (\texttt{afvr}\ (\texttt{avr}\ x\ t))}$$

(a) `aterm_afvr`

$$\frac{\texttt{aterm}\ t_1 \qquad \texttt{aterm}\ A \qquad \forall x \notin L.\texttt{aterm}\ t_2^{\texttt{avr}\ x\ A}}{\texttt{aterm}\ (\texttt{abnd}\ b\ t_1\ t_2)}$$

(b) `aterm_bnd`

Figure 2: Closure of type annotated terms.

One may wonder why we use an arbitrary type label A for x instead of the more obvious choice t_1, because in any reasonable type system the type of x is t_1. Choosing t_1 results in problems while trying to prove `subst_term` because the type label is changed in the goal but not in the induction hypothesis.

$$\frac{\cdots \qquad \cdots \qquad H : \texttt{aterm}\ [v \to u]t_2^{\texttt{avr}\ x\ t_1}}{\texttt{aterm}\ [v \to u]t_2^{\texttt{avr}\ x\ ([v \to u]t_1)}}$$

This would most likely require the notion of hereditary substitution. Furthermore this approach is troublesome for the definition of β-reduction because [9] requires that β-reduction should never take place in labels. Moreover one could try to represent `aterm_bnd` as follows.

$$\frac{\texttt{aterm}\ t_1 \qquad \forall(\texttt{avr}\ x\ A) \notin L.\texttt{aterm}\ A \implies \texttt{aterm}\ t_2^{\texttt{avr}\ x\ A}}{\texttt{aterm}\ (\texttt{abnd}\ b\ t_1\ t_2)}$$

This definition gives a stronger induction hypothesis, but non strictly positive occurrences of an inductive type are not allowed in Coq.

2.2 β-reduction and equality

In this section we define the notion of β-reduction on usual terms and type annotated terms, this notion is required for the conversion rule in PTSs and Γ_∞. To reduce the number of cases we have to consider in case analyses we use non-overlapping β-reduction [13] instead of ordinary β-reduction.

Definition 2.25. *A term* t_1 reduces in one non-overlapping step *to* t_2 *if* $t_1 \rightarrow_p t_2$ *(formally* `pbeta`*), where* $\rightarrow_p$ *is defined by the rules in Figure 3.*

$$\frac{\texttt{pterm}\ (\texttt{pbnd}\ \texttt{abs}\ t_1\ t_2) \qquad \texttt{pterm}\ u}{\texttt{papp}\ (\texttt{pbnd}\ \texttt{abs}\ t_1\ t_2)\ u \rightarrow_p t^u}$$

(a) `pbeta_red`

$$\frac{t_1 \rightarrow_p t_1' \qquad t_2 \rightarrow_p t_2'}{\texttt{papp}\ t_1\ t_2 \rightarrow_p \texttt{papp}\ t_1'\ t_2'}$$

(b) `pbeta_app`

$$\frac{t_1 \rightarrow_p t_1' \qquad \forall x \notin L. t_2^x \rightarrow_p t_2'^x}{\texttt{pbnd}\ b\ t_1\ t_2 \rightarrow_p \texttt{pbnd}\ b\ t_1'\ t_2'}$$

(c) `pbeta_bnd`

Figure 3: Non-overlapping β-reduction on usual lambda terms.

β-reduction is defined such that only closed terms participate in reduction, hence we are able to prove the following lemma.

Lemma 2.26. `pbeta_pterm` : $t_1 \rightarrow_p t_2 \implies \texttt{pterm}\ t_1 \wedge \texttt{pterm}\ t_2$

Lemma 2.27. `psubst_pbeta` : $t_1 \rightarrow_p t_2 \wedge \texttt{pterm}\ u \implies [x \rightarrow u]t_1 \rightarrow_p [x \rightarrow u]t_2$

The notion of β-reduction on type annotated terms is defined nearly the same as on usual terms, except for the binder case. Note that this definition forbids β-reduction to take place in type labels, as required by [9].

Definition 2.28. *A type annotated term* t_1 reduces in one non-overlapping step *to* t_2 *if* $t_1 \rightarrow_a t_2$ *(formally* `abeta`*), where* $\rightarrow_a$ *is a modification of* $\rightarrow_p$*. The relevant changes are shown in Figure 4.*

$$\frac{t_1 \rightarrow_a t_1' \qquad \texttt{aterm}\ A \qquad \forall x \notin L. t_2^{\texttt{avr}\ x\ A} \rightarrow_a t_2'^{\texttt{avr}\ x\ A}}{\texttt{abnd}\ b\ t_1\ t_2 \rightarrow_a \texttt{abnd}\ b\ t_1'\ t_2'}$$

(a) `abeta_bnd`

Figure 4: Non-overlapping β-reduction on type annotated terms.

Note that it is possible to use t_1 or t_1' as type label for t_2, but to remain consistent with `aterm` we use an arbitrary type label A.

Lemma 2.29. `abeta_aterm` : $t_1 \rightarrow_a t_2 \implies \texttt{aterm}\ t_1 \wedge \texttt{aterm}\ t_2$

Lemma 2.30. `asubst_abeta` : $t_1 \rightarrow_a t_2 \wedge \texttt{aterm}\ u \implies [x \rightarrow u]t_1 \rightarrow_a [x \rightarrow u]t_2$

Definition 2.31. *The reflexive symmetric transitive closure* $\approx_{(R,S)}$ *(formally* `equiv`*) of a binary relation* R *and a predicate* S *such that* $R\ t_1\ t_2 \implies S\ t_1 \wedge S\ t_2$ *is defined by the rules in Figure 5.*

$$\frac{S\ t}{t \approx_{(R,S)} t}$$
(a) `equiv_refl`

$$\frac{t_1 \approx_{(R,S)} t_2}{t_2 \approx_{(R,S)} t_1}$$
(b) `equiv_sym`

$$\frac{t_1 \approx_{(R,S)} t_2 \qquad t_2 \approx_{(R,S)} t_3}{t_1 \approx_{(R,S)} t_3}$$
(c) `equiv_trans`

$$\frac{R\ t_1\ t_2}{t_1 \approx_{(R,S)} t_2}$$
(d) `equiv_step`

Figure 5: Reflexive symmetric transitive closure.

Now we can define β-equivalence $\approx_p$ on usual lambda terms as $\approx_{(\texttt{pterm},\texttt{pbeta})}$ and β-equivalence $\approx_a$ on type annotated terms as $\approx_{(\texttt{aterm},\texttt{abeta})}$. Since only closed terms participate in our definitions of β-reduction this is well-defined. The following important lemmas are now trivial to prove.

Lemma 2.32. `pbeta_equiv_pterm` : $t_1 \approx_p t_2 \implies \texttt{pterm}\ t_1 \wedge \texttt{pterm}\ t_2$

Lemma 2.33. `abeta_equiv_aterm` : $t_1 \approx_a t_2 \implies \texttt{aterm}\ t_1 \wedge \texttt{aterm}\ t_2$

3 Translation between terms

In this section we discuss the operations used to translate between terms of both systems, these operations are necessary for the correspondence theorems discussed in Section 6. First we define the translation from type annotated terms to usual terms, this operation is informally described by erasure of type labels.

Definition 3.1. $|M|$ *(formally* `atp`*) is* the translation *of M to a usual term.*

$$\begin{aligned}
|\texttt{asrt}\ s| &= \texttt{psrt}\ s \\
|\texttt{abvr}\ n| &= \texttt{pbvr}\ n \\
|\texttt{afvr}\ (\texttt{avr}\ x\ a)| &= \texttt{pfvr}\ x \\
|\texttt{abnd}\ b\ t_1\ t_2| &= \texttt{pbnd}\ b\ |t_1|\ |t_2| \\
|\texttt{aapp}\ t_1\ t_2| &= \texttt{papp}\ |t_1|\ |t_2|
\end{aligned}$$

The following lemmas are proven in the formal development.

Lemma 3.2. `atp_open` : $|t^u| = |t|^{|u|}$

Lemma 3.3. `aterm_imp_pterm` : $\texttt{aterm}\ t \implies \texttt{pterm}\ |t|$

Lemma 3.4. `abeta_imp_pbeta` : $t_1 \rightarrow_a t_2 \implies |t_1| \rightarrow_p |t_2|$

Lemma 3.5. `abeta_imp_pbeta_equiv` : $t_1 \cong_a t_2 \implies |t_1| \cong_p |t_2|$

Before we are able to define the translation in the other direction we need the notion of environments to bind variables to terms (or rather types).

Definition 3.6. *An* environment `env`*, usually written as Γ, is a finite association list.*

$$\texttt{env} ::= \texttt{list}\ (\texttt{var} \times \texttt{ptrm})$$

Pairs are denoted as $x : A$, besides we have the obvious operations on contexts like $x : A \in \Gamma$ and `dom` Γ*.*

At first sight this translation looks easy: label the free variables with the corresponding types in the environment. It would be convenient to define this operation as proposed in [9].

Proposition 3.7. *M_Γ is the* type annotated term *of M in Γ.*

$$\begin{aligned}
(\texttt{psrt}\ s)_\Gamma &= \texttt{asrt}\ s \\
(\texttt{pbvr}\ n)_\Gamma &= \texttt{abvr}\ n \\
(\texttt{pfvr}\ x)_\Gamma &= \texttt{afvr}\ (\texttt{avr}\ x\ A_\Gamma) \qquad \textit{if}\ x : A \in \Gamma \\
(\texttt{pbnd}\ b\ t_1\ t_2)_\Gamma &= \texttt{abnd}\ b\ (t_1)_\Gamma\ (t_2)_\Gamma \\
(\texttt{papp}\ t_1\ t_2)_\Gamma &= \texttt{aapp}\ (t_1)_\Gamma\ (t_2)_\Gamma
\end{aligned}$$

Unfortunately we are not allowed to define it like this in `Coq` (and any terminating theory) because this function has the following "bad" properties.

1. The result is undefined if a free variable does not occur in the context.
2. It may "loop" if the context is ill-defined (e.g. $\Gamma = A : B, B : A$).

Therefore we need to find another way to define this function, we will discuss several unsuccessful approaches first.

1. We tried defining the operation by mutual induction on the structure of the context and the pseudo-term and let it yield an `option` type. The mutual induction ensures that the function terminates since the length of the environment decreases. Unfortunately mutual induction and `option` types are inconvenient to reason with. The first is due the non-mutual inductive definition of a PTS, the latter because an `option` type results in many extra case analyses.
2. We tried to construct M_Γ and A_Γ simultaneously by induction on the structure of the derivation of $\Gamma \vdash M : A$. This method has the following problems.
   (a) Consider the ordinary definition of the λ rule (with side conditions omitted).
   $$\frac{\Gamma, x : A \vdash M^x : B^x \qquad \Gamma \vdash \Pi A.B : s}{\Gamma \vdash \lambda A.M : \Pi A.B}$$
   In order to construct $(\lambda A.M)_\Gamma$ and $(\Pi A.B)_\Gamma$ we need to obtain A_Γ. By structural induction we only know $(\Pi A.B)_\Gamma$, but we have no clue about its shape, hence we cannot obtain A_Γ. Therefore we need to modify the λ rule to a well-known equivalent [5].
   $$\frac{\Gamma, x : A \vdash M^x : B^x \qquad \Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B^x : s_2}{\Gamma \vdash \lambda A.M : \Pi A.B}$$
   (b) Lemmas about β-conversion, like $t_1 \to_p t_2 \implies (t_1)_\Gamma \to_a (t_2)_\Gamma$ depend on $\Gamma \vdash M : A$. Not only is it hard to prove this lemma, it also requires 3 versions of the same lemma. Take $\Gamma \vdash A : B$ and $\Gamma \vdash M : N$, then we need to prove it for $A \to_p M$, $B \to_p N$ and $A \to_p N$ (we can omit $M \to_p B$ by symmetry).

(c) It is very hard to reason about the shape of the results, for example a trivial lemma like $(\texttt{psrt}\ s)_\Gamma = \texttt{asrt}\ s$ is hard to prove (especially because inversion on functions is not implemented in Coq).

3. We can reformulate Proposition 3.7 as a relation. But this function is defined on the structure of pseudo-terms instead of the structure of closed terms. Hence it does not give sufficiently strong induction hypotheses.

Therefore we have to define this translation in the same way as we have defined closed terms (Section 2.1) and β-reduction (Section 2.2).

Definition 3.8. *A pseudo-term* $t \in \texttt{ptrm}$ translates *to* $t' \in \texttt{atrm}$ *in* Γ *iff* $t \rightsquigarrow_\Gamma t'$, *where* `pta` *is mutually defined with* `pta_wde` *by the rules in Figure 6.*

$$\frac{\texttt{pta\_wde}\ \Gamma}{\texttt{psrt}\ s \rightsquigarrow_\Gamma \texttt{asrt}\ s}$$

(a) `pta_srt`

$$\frac{\texttt{pta\_wde}\ \Gamma \qquad A \rightsquigarrow_\Gamma A'}{x \rightsquigarrow_\Gamma \texttt{avr}\ x\ A}\ x : A \in \Gamma$$

(b) `pta_fvr`

$$\frac{t_1 \rightsquigarrow_\Gamma t_1' \qquad \forall x \notin L . t_2^x \rightsquigarrow_{(\Gamma, x:\texttt{psrt}\ s)} t_2'^{\texttt{avr}\ x\ (\texttt{asrt}\ s)}}{\texttt{pbnd}\ b\ t_1\ t_2 \rightsquigarrow_\Gamma \texttt{pbnd}\ b\ t_1'\ t_2'}$$

(c) `pta_bnd`

$$\frac{t_1 \rightsquigarrow_\Gamma t_1' \qquad t_2 \rightsquigarrow_\Gamma t_2'}{\texttt{papp}\ t_1\ t_2 \rightsquigarrow_\Gamma \texttt{aapp}\ t_1'\ t_2'}$$

(d) `pta_app`

$$\frac{}{\texttt{pta\_wde nil}}$$

(e) `pta_wde_nil`

$$\frac{A \rightsquigarrow_\Gamma A' \qquad \texttt{pta\_wde}\ \Gamma}{\texttt{pta\_wde}\ (\Gamma, x : A)}\ x \notin \texttt{dom}\ \Gamma$$

(f) `pta_wde_cons`

Figure 6: Translation from usual lambda terms to type annotated terms.

Again this notion is defined in such way that only closed terms and valid environments participate in the translation. Hence we are able to prove the following lemmas.

Lemma 3.9. $\texttt{pta\_term} : t \rightsquigarrow_\Gamma t' \implies \texttt{pterm}\ t \wedge \texttt{aterm}\ t'$

Lemma 3.10. $\texttt{pta\_wd\_env} : t \rightsquigarrow_\Gamma t' \implies \texttt{pta\_wde}\ \Gamma$

Now we should prove that this relation behaves as a function, i.e. its result is uniquely defined and exists if certain preconditions hold. First notice that in the definition of `pta_bnd` the type $(\texttt{asrt}\ s)$ is used as the label of x instead of an arbitrary type. If we use an arbitrary label we need a renaming lemma to prove `pta_unique`, but our renaming lemma (Lemma 3.13) depends on `pta_unique`.

Lemma 3.11. $\texttt{pta\_unique} : t \rightsquigarrow_\Gamma t_1 \wedge t \rightsquigarrow_\Gamma t_2 \implies t_1 = t_2$

Lemma 3.12. `pta_exists`

$$\texttt{pterm}\ t\ \wedge\ \texttt{fv}\ t \subseteq \texttt{dom}\ \Gamma\ \wedge\ \texttt{pta\_wde}\ \Gamma \implies \exists t'.t \rightsquigarrow_\Gamma t'$$

In our proof of `pta_exists` we ended up with a goal of the following shape.

$$\frac{\cdots \qquad t_1 \rightsquigarrow_\Gamma t_1' \qquad H : \forall x \notin L\ .\ \forall \Gamma\ .\ \ldots \implies \exists t_2'.t_2^x \rightsquigarrow_\Gamma t_2'}{\exists t'.\texttt{pbnd}\ b\ t_1\ t_2 \rightsquigarrow_\Gamma t'}$$

An obvious choice for the required witness is ($\texttt{abnd}\ b\ t_1'\ t_2'$), where t_2' could be obtained by instantiating H with a fresh variable x and using `shape_pterm` afterwards. But now we have to show that $\texttt{pbnd}\ b\ t_1\ t_2 \rightsquigarrow_\Gamma \texttt{abnd}\ b\ t_1'\ t_2'$, unfortunately `aterm_bnd` is universally quantified which means this yields an universally quantified goal. Sadly we only know it holds just for our freshly taken x, therefore we need the following renaming lemma.

Lemma 3.13. `pta_rename` : $\texttt{A} \rightsquigarrow_\Gamma \texttt{A}' \wedge \texttt{B} \rightsquigarrow_\Gamma \texttt{B}' \wedge \texttt{y} \notin \texttt{dom E} \wedge \texttt{x} \notin \texttt{fv}\ \texttt{t}_1 \wedge (\texttt{avr x A}) \notin \texttt{afva}\ \texttt{t}_2$

$$t_1^x \rightsquigarrow_{(\Gamma,x:A)} t_2^{\texttt{avr}\ x\ A'} \implies t_1^y \rightsquigarrow_{(\Gamma,y:B)} t_2^{\texttt{avr}\ y\ B'}$$

For the proof of `pta_rename` we need a substitution lemma, but first we require the weakening lemma and the notion of substitution on environments and hereditary free variable substitution.

Lemma 3.14. `pta_weaken` : `pta_wde` (Γ, Σ, Δ)

$$t \rightsquigarrow_{(\Gamma,\Delta)} t' \implies t \rightsquigarrow_{(\Gamma,\Sigma,\Delta)} t'$$

Definition 3.15. Hereditary substitution *of a free variable x for u in t (formally* `hsubst`*) is defined as follows.*

$$\begin{aligned}
[x \rightarrow_h u]\ \texttt{asrt}\ s &= \texttt{srt}\ s \\
[x \rightarrow_h u]\ \texttt{abvr}\ i &= \texttt{bvr}\ i \\
[x \rightarrow_h u]\ \texttt{afvr}\ (\texttt{avr}\ y\ B) &= \texttt{if}\ (\texttt{avr}\ y\ B = y)\ \texttt{then}\ u \\
&\qquad \texttt{else}\ (\texttt{afvr}\ (\texttt{avr}\ y\ [x \rightarrow_h u]B)) \\
[x \rightarrow_h u]\ \texttt{abnd}\ b\ t_1\ t_2 &= \texttt{abnd}\ b\ [x \rightarrow_h u]t_1\ [x \rightarrow_h u]t_2 \\
[x \rightarrow_h u]\ \texttt{aapp}\ t_1\ t_2 &= \texttt{aapp}\ [x \rightarrow_h u]t_1\ [x \rightarrow_h u]t_2
\end{aligned}$$

Lemma 3.16. `pta_subst` : $u \rightsquigarrow_\Gamma u' \wedge A \rightsquigarrow_\Gamma A'$

$$t \rightsquigarrow_{(\Gamma,x:A,\Delta)} t' \implies [x \rightarrow u]t \rightsquigarrow_{(\Gamma,[x\rightarrow u]\Delta)} [\texttt{avr}\ x\ A' \rightarrow_h u']t'$$

Note that non-hereditary substitution does not work because substitution takes place in the environment and hence may change the type labels. For example non-hereditary substitution of A for s in $y \rightsquigarrow_{(A:s,y:A)} y^A$ results in $y \rightsquigarrow_{(y:s)} y^A$ which is certainly not correct.

Now we are able to prove the following lemmas which state that β-reduction and β-equality are preserved under translation. We need these lemmas for the correspondence theorems in Section 6.

Lemma 3.17. `pbeta_imp_abeta`

$$t_1 \rightarrow_p t_2 \ \wedge\ t_1 \rightsquigarrow_\Gamma t_1' \ \wedge\ t_2 \rightsquigarrow_\Gamma t_2' \implies t_1' \rightarrow_a t_2'$$

Lemma 3.18. `pbeta_imp_abeta_equiv`

$$t_1 \approx_p t_2 \ \wedge\ t_1 \rightsquigarrow_\Gamma t_1' \ \wedge\ t_2 \rightsquigarrow_\Gamma t_2' \implies t_1' \approx_a t_2'$$

The proof of `pbeta_imp_abeta_equiv` contains a tricky part in the transitivity case. Consider the following diagram.

$$\begin{tikzcd}
t_1 \arrow[r, dash, "\approx_p"] \arrow[d, squiggly, "\Gamma"] & t_2 \arrow[r, dash, "\approx_p"] \arrow[d, dotted, "{(\Gamma,\Sigma)}"] & t_3 \arrow[d, squiggly, "\Gamma"] \\
t_1' \arrow[r, dotted, dash, "\approx_a"] & \textbf{?} \arrow[r, dotted, dash, "\approx_a"] & t_3'
\end{tikzcd}$$

Here we are supposed to specify a witness for **?** such that the diagram commutes, but there may be free variables in t_2 that are bound by Γ, hence `pta_exists` does not yield the required witness. We solve this by widening Γ to (Γ, Σ) where $\texttt{dom}\ \Sigma = \texttt{fv}\ t_2 \backslash \texttt{dom}\ \Gamma$ such that each $x \in \texttt{dom}\ \Sigma$ has some bogus type assigned.

Lemma 3.19. `pta_wde_widen`

$$\forall L\ .\ \texttt{pta\_wde}\ \Gamma \implies \exists \Sigma\ .\ L \subseteq \texttt{dom}\ (\Gamma, \Sigma)\ \wedge\ \texttt{pta\_wde}\ (\Gamma, \Sigma)$$

4 Pure Type Systems

In this section we present our representation of Pure Type Systems. We only describe lemmas required for our developments, for an extensive discussion of PTSs and their properties we refer to [3] and for a more substantial formalization of PTSs we refer to [13, 1].

Definition 4.1. *A* Pure Type System (PTS) *is s system with derivation rules given in Figure 7 (formally* `pts`*) and is parametrized by the following relations:*

1. *Axioms,* $\texttt{axiom} \subseteq \texttt{sort} \times \texttt{sort}$
2. *Rules,* $\texttt{rule} \subseteq \texttt{sort} \times \texttt{sort} \times \texttt{sort}$

Because a PTS is defined in the same way as closed terms it automatically follows that any term, type, and type in its context is closed.

Lemma 4.2. `pts_imp_pterm`

$$\Gamma \vdash M : A \implies \texttt{pterm}\ M\ \wedge\ \texttt{pterm}\ A\ \wedge\ \forall y : C \in \Gamma\ .\ \texttt{pterm}\ C$$

Lemma 4.3. $\texttt{pts\_fv} : \Gamma \vdash M : B$

$$(\exists y : C \in \Gamma\ .\ x \in \texttt{fv}\ C)\ \vee\ x \in \texttt{fv}\ M\ \vee\ x \in \texttt{fv}\ B \implies x \in \texttt{dom}\ \Gamma$$

5 Pure Type Systems à la Γ_∞

In this section we present our representation of Pure Type Systems à la Γ_∞.

Definition 5.1. *A* Pure Type System (PTS) à la Γ_∞ *is a system with derivation rules given in Figure 8 (formally* `ginf`*) and parametrized by the relations* `axiom` *and* `rule`*.*

$$\frac{}{\vdash \texttt{psrt}\ s_1 : \texttt{psrt}\ s_2}\ \texttt{axiom}\ s_1\ s_2$$
(a) pts_srt

$$\frac{\Gamma \vdash A : \texttt{psrt}\ s}{\Gamma, x : A \vdash \texttt{pfvr}\ x : A}\ x \notin \Gamma$$
(b) pts_var

$$\frac{\Gamma \vdash A : \texttt{psrt}\ s \qquad \Gamma \vdash M : C}{\Gamma, x : A \vdash M : C}\ x \notin \Gamma$$
(c) pts_weak

$$\frac{\Gamma \vdash A : \texttt{psrt}\ s_1 \qquad \forall x \notin L. \Gamma, x : A \vdash B^x : \texttt{psrt}\ s_2}{\Gamma \vdash \texttt{ppi}\ A\ B : \texttt{psrt}\ s_3}\ \texttt{rule}\ s_1\ s_2\ s_3$$
(d) pts_pi

$$\frac{\forall x \notin L. \Gamma, x : A \vdash M^x : B^x \qquad \Gamma \vdash \texttt{ppi}\ A\ B : \texttt{psrt}\ s}{\Gamma \vdash \texttt{pabs}\ A\ M : \texttt{ppi}\ A\ B}$$
(e) pts_abs

$$\frac{\Gamma \vdash M : \texttt{ppi}\ A\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash \texttt{papp}\ M\ N : B^N}$$
(f) pts_app

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \texttt{psrt}\ s}{\Gamma \vdash M : B}\ A \approx_p B$$
(g) pts_conv

Figure 7: The deduction rules of a PTS

Now one might wonder whether our definition using co-finite quantification corresponds to the original definition given in [9]. To give the reader more confidence we have also stated a definition using the exists-fresh approach. The Π rule of this definition is displayed below, the λ rule is defined similarly.

$$\frac{A :_\infty \texttt{asrt}\ s_1 \qquad (\texttt{avr}\ x\ A) \notin \texttt{ahfv}\ B \qquad B^{\texttt{avr}\ x\ A} :_\infty \texttt{asrt}\ s_2}{\texttt{api}\ A\ B :_\infty \texttt{asrt}\ s_3}\ \texttt{rule}\ s_1\ s_2\ s_3$$

Moreover we have proven that derivations in this exists-fresh version of Γ_∞ correspond exactly to derivations in the co-finite version. The proof of one direction is trivial but for the other direction we need a renaming lemma which is stated below.

Lemma 5.2. ginf_imp_aterm : $M :_\infty A \implies \texttt{aterm}\ M \ \wedge\ \texttt{aterm}\ A$

Lemma 5.3. ginf_ahfv_well_typed

$$(\texttt{avr}\ x\ A) \in \texttt{ahfv}\ M \cup \texttt{ahfv}\ C \ \wedge\ M :_\infty C \implies \exists s. A :_\infty \texttt{asrt}\ s$$

Lemma 5.4. ginf_hsubst

$$M :_\infty B \ \wedge\ N :_\infty A \implies [\texttt{avr}\ x\ A \to_h N]M :_\infty [\texttt{avr}\ x\ A \to_h N]B$$

Lemma 5.5. ginf_rename : $\texttt{avr}\ x\ A \notin \texttt{ahfv}\ M \cup \texttt{ahfv}\ B \wedge \texttt{aterm}\ A$

$$M^{\texttt{afvr}\ (\texttt{avr}\ x\ A)} :_\infty B^{\texttt{afvr}\ (\texttt{avr}\ x\ A)} \implies M^{\texttt{afvr}\ (\texttt{avr}\ y\ A)} :_\infty B^{\texttt{afvr}\ (\texttt{avr}\ y\ A)}$$

Note that non-hereditary substitution does not work here either, for example $x^{y^{s_1}} :_\infty y^{s_1}$ is derivable in Γ_∞, but if we substitute s_1 for y^{s_1} we obtain nonsense like $x^{y^{s_1}} :_\infty s_1$. It is also remarkable that this lemma is much easier to prove than the equivalent lemma for PTSs. This is obviously caused because Γ_∞ does not have contexts and therefore we do not have to worry about weakening.

$$\frac{}{\texttt{asrt}\ s_1 :_\infty \texttt{asrt}\ s_2}\ \texttt{axiom}\ s_1\ s_2$$

(a) `ginf_srt`

$$\frac{A :_\infty \texttt{asrt}\ s}{\texttt{afvr}\ (\texttt{avr}\ x\ A) :_\infty A}$$

(b) `ginf_var`

$$\frac{A :_\infty \texttt{asrt}\ s_1 \qquad \forall x \notin L. B^{\texttt{avr}\ x\ A} :_\infty \texttt{asrt}\ s_2}{\texttt{api}\ A\ B :_\infty \texttt{asrt}\ s_3}\ \texttt{rule}\ s_1\ s_2\ s_3$$

(c) `ginf_pi`

$$\frac{\forall x \notin L. M^x :_\infty B^x \qquad \texttt{api}\ A\ B :_\infty \texttt{asrt}\ s}{\texttt{aabs}\ A\ M :_\infty \texttt{api}\ A\ B}$$

(d) `ginf_abs`

$$\frac{M :_\infty \texttt{api}\ A\ B \qquad N :_\infty A}{\texttt{aapp}\ M\ N :_\infty B^N}$$

(e) `ginf_app`

$$\frac{M :_\infty A \qquad B :_\infty \texttt{asrt}\ s}{M :_\infty B}\ A \approx_a B$$

(f) `ginf_conv`

Figure 8: The deduction rules of a PTS in Γ_∞-style

6 The correspondence theorem

Now we have defined the necesarry infrastructure and have proven the required lemmas about PTSs and Γ_∞ we can state one direction of the correspondence theorem between PTSs and Γ_∞.

Theorem 6.1. `pts_exists_ginf`

$$\Gamma \vdash M : A \implies \exists M' \exists A'\ .\ M \rightsquigarrow_\Gamma M' \ \wedge\ A \rightsquigarrow_\Gamma A' \ \wedge\ M' :_\infty A'$$

We have used the following lemmas to prove `pts_exists_ginf`.

Lemma 6.2. `pts_pta_wde` : $\Gamma \vdash M : A \implies$ `pta_wde` Γ

Lemma 6.3. `pts_imp_ginf`

$$\Gamma \vdash M : A \ \wedge\ M \rightsquigarrow_\Gamma M' \ \wedge\ A \rightsquigarrow_\Gamma A' \implies M' :_\infty A'$$

The proof of the other direction of this lemma is unfortunately absent in this development.

7 Conclusions and further research

In this paper we have discussed our formalization of Γ_∞ in Coq, this led to various insights which we discuss in this section.

Dealing with two languages that only differ with respect to their free variables is an important part of this development. We have discussed that creating separate definitions for each language is undesirable because it results in duplication of definitions and lemmas. This problem is not discussed in previous work such as [1, 13, 2, 5] (simply because they did not need it), hence we developed a method ourselves.

We have discussed (and dismissed) various methods to realize this goal. Nonetheless our definite method suffers from many implementation issues; Coq is unable to derive induction schemes automatically, it is not possible to define "nested" recursive functions easily and we were unable to construct an abstract type representing finite sets of variables. Thus we still had to do much work two times, so there is definitely further research required to improve this methodology.

Co-finite quantification presented by Aydemir *et al.* [2] worked quite well for our developments. We tried exists-fresh approaches as well but rapidly ended up with the same issues as described in [2]; the induction hypothesis are not strong enough to prove various lemmas in an easy way. We also have encountered some problems with Aydemir's methodology, namely to prove existentially quantified goals a renaming lemma is required.

It turned out that the framework of Aydemir *et al.* [2] was less suitable for our developments due to our notion of type annotated variables. In order to make an implementation of type annotated variables in their framework's module `Var` we are required to define a total order on `avar`. This is an unnecessary burden so we stepped back to `Coq`'s library which supports weak finite sets. To avoid ending up using multiple implementations (and hence different named and stated lemmas) we have used `Coq`'s finite set library for both kind of finite sets. Nonetheless the formalization of CoC in Aydemir's framework was very useful to us. Many concepts, including some `Coq` tactics, could easily be reused for our developments.

Some theorems that are very hard to prove for ordinary PTSs are rather easy to prove in Γ_∞-style. Since contexts are absent, lemmas like thinning, weakening and strengthening are trivial. Therefore lemmas about substitution and renaming are much easier to prove. Hence it might be useful to transform a given theorem to Γ_∞, prove it there, and then translate it back.

An obvious goal for further research is to extend this formalization to the other direction of the correspondence theorem. The proof of this direction as given by Geuvers *et al.* requires a lot of work, mainly because it depends on the strengthening lemma. According to previous research [1, 13] this lemma is very hard to prove. Furthermore a lot of operations on contexts and the notion of topological sorting are required.

During our developments it has become clear that many details in [9] are are left implicit and our formalization has definitely cleared these up. We conclude that formalizing of (meta) mathematics is very useful!

Acknowledgements

The author would like to thank James McKinna for his supervision, Freek Wiedijk for his review, the participants of the Coq Club mailinglist for their kind answers to the author's questions and Brian Aydemir and Arthur Charguéraud for making their framework and formalizations publicly available.

References

[1] R. Adams. A Formalization of the Theory of Pure Type Systems in Coq. Obtained from `http://www.cs.rhul.ac.uk/~robin/coqPTS/` on Decem-

ber 5, 2009.

[2] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S Weirich. Engineering Formal Metatheory. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, January 2008.

[3] H. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[4] B. Barras. Coq en coq. Rapport de Recherche 3026, INRIA, 1996.

[5] L.S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking Algorithms for Pure Type Systems. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, pages 19–61. Springer-Verlag, 1994.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[7] N. G. De Bruijn. Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae*, pages 381–392, 1972.

[8] The Coq Development Team. The Coq Proof Assistant. Web page, obtained from `http://coq.inria.fr` on December 5, 2009.

[9] H. Geuvers, J. McKinna, and F. Wiedijk. Pure Type Systems without Explicit Contexts. Unpublished, obtained from `http://www.cs.ru.nl/~james/2009-TLCA/submitted.pdf` on December 5, 2009.

[10] J. Harrison. The HOL Light theorem prover. Web page, obtained from `http://www.cl.cam.ac.uk/~jrh13/hol-light/` on December 5, 2009.

[11] R Krebbers, A. Chlipala, B. Aydemir, and B. Barras. Meta theory: induction over terms with abstract variables. Thread on Coq Club mailinglist, `http://logical.saclay.inria.fr/coq-puma/messages/54ac6bcf64e2d60f`, 2009.

[12] C. McBride and J. McKinna. I Am Not a Number; I Am a Free Variable. In *Proceedings of ACM SIGPLAN Haskell Workshop*, 2004.

[13] J. McKinna and R. Pollack. Some Lambda Calculus and Type Theory Formalized. *Journal of Automated Reasoning*, 23(3–4), November 1999.

[14] The Mizar Development Team. Mizar Home Page. Web page, obtained from `http://mizar.uwb.edu.pl` on December 5, 2009.

[15] R. Pollack. The LEGO Proof Assistant. Web page, obtained from `http://www.dcs.ed.ac.uk/home/lego/` on December 5, 2009.

[16] R. Pollack. How to Believe a Machine-Checked Proof. In *Twenty Five Years of Constructive Type Theory.* Oxford Univ. Press, 1998.