

Pure Type Systems without Explicit Contexts

Herman Geuvers, James McKinna, and Freek Wiedijk

Institute for Computing and Information Sciences
Radboud University Nijmegen
Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands

Abstract. We present an approach to type theory in which the typing judgments do not have explicit contexts. Instead of judgments of the shape $\Gamma \vdash A : B$, our systems just have judgments of the shape $A : B$. An essential feature of our systems is that already in pseudo-terms we distinguish the free from the bound variables.

Specifically we give the rules of the ‘Pure Type System’ class of type theories in this style. We prove that the type judgments of these systems corresponds in a natural way with the judgments of the traditionally formulated Pure Type Systems. I.e., our systems have exactly the same well-typed terms as traditional presentations of type theory.

Our system can be seen as a type theory in which all type judgments share an identical, infinite, typing context that has infinitely many variables for each possible type. For this reason we call our system Γ_∞ . This name means to suggest that our type judgment $A : B$ should be read as $\Gamma_\infty \vdash A : B$, with a fixed infinite type context called Γ_∞ .

1 Introduction

1.1 Problem

One of the important insights type theory gave us is that one needs to be aware of the *context* in which one is working. This already was stressed in 1979 by de Bruijn in his paper *Wees contextbewust in WOT*, Dutch for ‘Be context aware in the mathematical vernacular’ [3]. In type theory a term always is considered in a context Γ , which gives the types of the free variables that occur in the term. This also is apparent in the typing judgments of type theory, which have the shape $\Gamma \vdash M : A$, where this context Γ is made explicit. In type theory a ‘bound’ variable is bound *locally* in a term, while a ‘free’ variable actually is *globally* bound, namely by the context of the term.

In the customary presentation of first order predicate logic, and in fact in the presentation of most other logics as well, free variables are not treated in such a way. In these logics free variables are really *free*. They are taken from an infinite supply of variables that are just available to be used in formulas and terms, without them having to be declared first.

This difference between type theory and predicate logic means that when we model predicate logic in type theory, actually we do not get the customary version of predicate logic, but instead get a version of predicate logic that is called

free logic instead [6]. In traditional treatments of predicate logic the formula $(\forall x. P(x)) \rightarrow (\exists x. P(x))$ usually is provable. For instance a natural deduction proof of this formula would look like:

$$\frac{\frac{\frac{[\forall x. P(x)]}{P(y)} \forall E}{\exists x. P(x)} \exists I}{(\forall x. P(x)) \rightarrow (\exists x. P(x))} \rightarrow I$$

In this proof one uses the free variable y . If one cannot use any other variables than the ones that are introduced by earlier rules, then this proof will not work. And in fact, in type theory the term that corresponds to this formula will not be inhabited. I.e., there is no term M such that

$$D : *, P : D \rightarrow * \vdash M : (\Pi x : D. P(x)) \rightarrow (\Sigma x : D. P(x))$$

because we cannot exclude that the domain D is empty, in which case the formula is false.

Now there are two things one can do to bridge this difference between type theory and traditional logic:

- Make predicate logic more like type theory, by explicitly keeping track in the judgments of the set of variables that can be used in the proof.
- Make type theory more like predicate logic, by having a version of type theory that does not need contexts in the judgments, i.e., in which free variables are just taken from an infinite supply of variables.

Although the first option is interesting too, especially in categorical treatments of logic [5, for example], in this paper we focus on the second. We originally thought that the dependent types in type theory would prevent a version of type theory without contexts from being a viable option, but to our surprise it turns out that one *can* present type theory in a style where there are no contexts and in which therefore free variables are really free, provided we are prepared to pay the small price of labelling variables in a rather involved manner.

There is another reason why it is interesting to look at a version of type theory where there are no explicit contexts. One of the most popular architectures for proof assistants is the *LCF architecture*, named after the LCF system from the seventies [8]. In such a system there is an abstract data type called `term`, that only can be created by a small number of functions exported from the type checking kernel. Elements of this datatype always correspond to type-correct terms.

It would be attractive to implement type theory according to this architecture. A system that uses this approach would have a kernel with an interface that has a function:

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

However, with the usual style type theories this approach is not attractive. The `app` function will need to check whether the contexts in which the terms live are compatible, which will be very expensive, if it needs to be done for the type checking of each function application. One should be aware that in a realistic system all available *definitions* are also part of the context. In a serious formalization there generally are hundreds of those definitions.

For this reason actual type theoretical proof assistant have a kernel that has a different kind of interface. In such a kernel there is no abstract datatype of *terms* (there just is a – non-abstract – type of *preterms*). Instead there is an abstract type of *contexts*, which we call `environment` here. The interface then looks like:

```
mkApp : preterm * preterm -> preterm
add_constant : string * preterm -> environment -> environment
```

(The names of these functions are the actual names that they have in the Coq system’s kernel. The types of those functions in the Coq system are essentially what is presented here.) The system then has a global variable:

```
global_env : environment ref
```

that corresponds to the *state* of the system in which the user of the system is working.

Although the architecture with the environments that we described is purely functional (as is the Coq kernel), the fact that the actual implementation has this global variable means that it is used in a rather ‘stateful’ way. The desire to investigate possibilities for an LCF-style kernel for type theory that is ‘less stateful’ motivated this research. In the conclusions we will address the question whether the style of type theory that we present here will lead to such a type checking architecture.

1.2 Approach

The approach that we will follow here is to imagine there to be an ‘infinite context’ called Γ_∞ . For each type correct type A this context will have infinitely many variables x_i^A . It should be stressed that this A should be considered to just be a label, a *string*. There will never be reduction happening inside these A s. Also, x_i^A and x_j^B will be different variables, even when A and B are convertible, or even if they are α -equivalent. Note that the variables in A themselves also will be of the shape x_j^B : this means that the terms will have a recursive tree-like structure.

For instance, a variable that corresponds to the successor function on natural numbers will be something like

$$x_0^{x_0^* \rightarrow x_0^*}$$

in which x_0^* is the variable for the natural numbers. If we use more suggestive names for the variables, this becomes:

$$s^{N^* \rightarrow N^*}$$

So the ‘small price’ alluded to above is that a free variable x_i^A in a well-typed term should carry with it the (well-founded) history of how it comes to be well-typed; in other words, that its labelling A does the work of a proof of the validity of the context extension $\Gamma, x_i : A$.

Now our systems will have judgments $A : B$, which really should be interpreted as $\Gamma_\infty \vdash A : B$. For this reason we call the general approach to type theory that we introduce here ‘ Γ_∞ ’ (reusing the name of the context as the name of the system). Note that Γ_∞ is not a single system: each type theory will have a ‘ Γ_∞ -variant’.

The Γ_∞ approach has the essential feature that there are two different classes of variables. There are the variables that come from the Γ_∞ context (the ‘free’ variables), and then there is another kind of ‘bound’ variables. When thinking about our systems one might imagine de Bruijn indices for the bound variables, although the presentation that we will give here will use named variables for them as well.

Although we expect that many type theories have a Γ_∞ -variant, to keep things concrete we here will only present the class of type theories called Pure Type Systems in the Γ_∞ style. That way everything will be concrete, and that will allow us to prove a precise correspondence theorem between Pure Type Systems in the traditional style and our version of the Pure Type Systems.

One should note that in our system obviously any type will be inhabited. Therefore, if one wants to use our type theories through the Curry-Howard-de Bruijn isomorphism, one should keep track of the free variables that occur in the proof terms.

1.3 Related Work

To our surprise, we did not find any prior published work investigating the approach that we propose here.

Conor McBride (private communication) observed that Pollack’s LEGO implementation already supported the idea, and this idea was then used in the architecture of EPIGRAM 2. However, this approach has not been treated theoretically as we do here.

The explicit distinction between free and bound variables on a syntactic level already can be found in [7]. Various approaches to representing variables in terms are discussed in [11], where having named free variables and de Bruijn index bound variables is considered one of the best options for mechanisation. However, the motivation in that work to distinguish between free and bound variables is quite different from the motivation that we had here.

Elsewhere, Pollack considered presentations of type theory which separate the typing judgment from that for context well-formedness [9], although judgments are still made ‘in context’. This allows a subtle range of issues to be explored, especially with respect to closure under alpha-conversion, which we treat only informally here.

1.4 Contribution

We present a different approach to type theory, where free variables are not bound in a finite context but are taken to be really free. This approach is much closer to the way that logical systems usually are presented than the standard presentation of type theory.

We also prove our approach correct by proving two theorems that establish a straightforward correspondence between the standard presentation of type theory, and the variant that we present here.

1.5 Outline

The structure of the paper is as follows. In Section 2 we recall the PTS rules and some of its theory. In Section 3 we present the Γ_∞ -variant of the PTS rules, in which the judgments do not have contexts. In Section 4 we show that both systems correspond to each other in a natural way. In Section 5 we conclude with a prospectus for an implementation based on our variant of the PTS rules.

2 Pure Type Systems in the traditional style

Pure Type Systems (PTSs) are a generalization of many existing type systems and thus the class of PTSs contains various well-known systems, like system F and system $F\omega$, dependent type theory λP and the Calculus of Constructions.

Definition 2.1. For \mathcal{S} a set (the set of sorts), $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ (the set of axioms) and $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ (the set of rules), the Pure Type System $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is the typed λ -calculus with inference rules given in Figure 1 on page 6. In the rules, the expressions are taken from the set of pseudo-terms \mathcal{T} defined by

$$\mathcal{T} ::= s \mid \mathcal{V} \mid \Pi \mathcal{V} : \mathcal{T}. \mathcal{T} \mid \lambda \mathcal{V} : \mathcal{T}. \mathcal{T} \mid \mathcal{T} \mathcal{T}.$$

with \mathcal{V} producing the variable names. (We leave the specific variable names used unspecified at this point, as this does not matter as long as there is a countably infinite set of variables. However, below we will take a specific choice for it.)

The Γ are taken from the set of pseudo-contexts

$$x_1 : A_1, \dots, x_n : A_n$$

where the x_i are all different and $A_i \in \mathcal{T}$.

There is a lot of theory about PTSs and various systems have been studied in the context of PTSs. We do not give a complete overview but refer to [1, 2, 4] for details and explanation. Here we just give the results that we will use in the rest of the paper to prove the equivalence between a PTS and its stateless version.

(sort)	$\frac{}{\vdash s_1 : s_2}$	if $(s_1, s_2) \in \mathcal{A}$
(var)	$\frac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A}$	if $x \notin \Gamma$
(weak)	$\frac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x:A \vdash M : C}$	if $x \notin \Gamma$
(Π)	$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x:A \vdash B : s_2}{\Gamma \vdash \Pi x:A.B : s_3}$	if $(s_1, s_2, s_3) \in \mathcal{R}$
(λ)	$\frac{\Gamma, x:A \vdash M : B \quad \Gamma \vdash \Pi x:A.B : s}{\Gamma \vdash \lambda x:A.M : \Pi x:A.B}$	
(app)	$\frac{\Gamma \vdash M : \Pi x:A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$	
(conv)	$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$	$A =_\beta B$

Fig. 1. Typing rules for PTSs

Definition 2.2. *The pseudo-term A is called well-typed if there is a pseudo-context Γ and a pseudo-term B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable. A pseudo-context Γ is well-formed if there are pseudo-terms A and B such that $\Gamma \vdash A : B$ is derivable. The set of variables declared in a pseudo-context Γ is called the domain of Γ , $\text{dom}(\Gamma)$; if $x \in \text{dom}(\Gamma)$, then $\text{type}_\Gamma(x)$ denotes the ‘type’ that x is assigned to in Γ . So if $x : A \in \Gamma$, then $\text{type}_\Gamma(x) = A$. The expression $\text{type}(\Gamma)$ denotes the set of such ‘types’ that occur in Γ . The set of well-typed terms of $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is denoted by $\text{Term}(\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R}))$.*

We adopt the usual notions of bound and free variable, α -conversion (\equiv), β -reduction (\rightarrow_β) and β -equality ($=_\beta$) on pseudo-terms. The conversion rule has as a side condition that $A =_\beta B$, i.e. A and B should be β -equal as pseudo-terms.

The following are two well-known properties of PTSs. The relation $\Gamma \subseteq \Delta$ just denotes the set containment between Γ and Δ as sets of variable assignments. The third property, Permutation, is a corollary of Strengthening.

Proposition 2.3. – **Thinning** *If $\Gamma \vdash M : A$ and $\Delta \supseteq \Gamma$ is a well-formed context, then $\Delta \vdash M : A$.*

– **Strengthening** *If $\Gamma, x : B, \Delta \vdash M : A$ and $x \notin \text{FV}(\text{type}(\Delta, M, A))$, then $\Gamma, \Delta \vdash M : A$.*

– **Permutation** *If $\Gamma, x : B, y : C, \Delta \vdash M : A$ and $x \notin \text{FV}(C)$, then $\Gamma, y : C, x : B, \Delta \vdash M : A$.*

In proving the equivalence between a PTS and its stateless version, we need to merge two well-formed contexts to create a new one. Therefore we introduce some new definitions.

Definition 2.4. *Let Γ and Δ be two pseudo-contexts. We say that Γ and Δ are compatible, notation $\Gamma \parallel \Delta$ if*

$$\forall x \in \text{dom}(\Gamma) \cap \text{dom}(\Delta) (\text{type}_\Gamma(x) \equiv \text{type}_\Delta(x)).$$

The merge of Γ and Δ , notation $\Gamma \times \Delta$ is the pseudo-context $\Gamma, (\Delta \setminus \Gamma)$. This is Γ followed by the declarations $x : B \in \Delta$ for which $x \notin \text{dom}(\Gamma)$.

Note the strong requirement in $\Gamma \parallel \Delta$ that the types of x in Γ and Δ should be α -equal, and not just β -convertible. The reason is that we want to merge compatible contexts and this may not be possible with the weaker requirement asking types to be β -equal. A counterexample is:

$$\begin{aligned} \Gamma &:= A : *, x : A, y : (\lambda v:A.A)x \\ \Delta &:= A : *, y : A, x : (\lambda v:A.A)y \end{aligned}$$

Γ and Δ cannot be merged into one well-formed context.

Lemma 2.5. *If Γ and Δ are well-formed contexts and $\Gamma \parallel \Delta$, then $\Gamma \times \Delta$ is well-formed.*

Proof. We write $x_1 : B_1, \dots, x_n : B_n$ for Δ . $\Gamma \times \Delta$ is the pseudo-context $\Gamma, (\Delta \setminus \Gamma)$. As Γ is well-formed, we only have to consider the part $\Delta \setminus \Gamma = x_{i_1} : B_{i_1}, \dots, x_{i_n} : B_{i_n}$.

$x_1 : B_1, \dots, x_{i_1-1} : B_{i_1-1} \subseteq \Gamma$, so by Thinning $\Gamma \vdash B_{i_1} : s$ for some sort s , so $\Gamma, x_{i_1} : B_{i_1}$ is well-formed.

The same reasoning applies to $x_{i-2} : B_{i-2}, \dots, x_{i-n} : B_{i-n}$, so we conclude that $\Gamma, (\Delta \setminus \Gamma)$ is well-formed.

3 Pure Type Systems in the Γ_∞ style

We begin by defining the pseudo-terms of Γ_∞ . These are straight-forward, apart from the fact that we need *two* kinds of variables: free variables x_i^A and bound variables x_i . (The bound variables really can be thought of as de Bruijn indices, but that should not matter; we do it the named way here.) So the grammar of these new pseudo-terms is:

$$\mathcal{T} ::= s \mid x_i \mid x_i^{\mathcal{T}} \mid \Pi x_i:\mathcal{T}. \mathcal{T} \mid \lambda x_i:\mathcal{T}. \mathcal{T} \mid \mathcal{T}\mathcal{T}$$

In this s is any sort of the system, i is a natural number, and x is just the letter x (so it does not vary over anything). We call this set \mathcal{T}_∞ .

It is important to note that the type labels in the free variables are just labels: we do not reduce inside them. We make this precise by defining β -equality.

Definition 3.1. We define β -reduction and β -conversion on \mathcal{T}_∞ as follows:

$$(\lambda x_i : A.M)N \rightarrow_\beta M[x_i := N]$$

If $M \rightarrow_\beta M'$, then $MP \rightarrow_\beta M'P$, $\lambda x_i : M.P \rightarrow_\beta \lambda x_i : M'.P$, $\lambda x_i : P.M \rightarrow_\beta \lambda x_i : P.M'$, $\Pi x_i : M.P \rightarrow_\beta \Pi x_i : M'.P$, $\Pi x_i : P.M \rightarrow_\beta \Pi x_i : P.M'$.

The β -convertibility is defined as the reflexive symmetric transitive closure of \rightarrow_β . To distinguish it from $=_\beta$, we denote it by \approx .

So $x_i^{(\lambda v : A.A)M} \not\approx x_i^A$.

We will now take a specific choice for the variable names in the PTS pseudo-terms (extending the grammar on page 5):

$$\mathcal{V} ::= x_i \mid x_i^{\mathcal{T}}$$

(Note that in this rule the \mathcal{T} ‘type labels’ are considered to be completely inert ‘strings’, that just *label* the variables.) If we take this choice then all terms from \mathcal{T}_∞ also will be terms of \mathcal{T} , so then we will have:

$$\mathcal{T}_\infty \subset \mathcal{T}$$

(In fact the \mathcal{T}_∞ pseudo-terms are exactly the pseudo-terms of \mathcal{T} in which all binders only use the ‘bound’ kind of variable name.)

One also could distinguish between \mathcal{T}_∞ and \mathcal{T} as syntactic categories, and talk about an *embedding* of \mathcal{T}_∞ into \mathcal{T} , but for notational simplicity we will not take that route here.

We have the following important property.

Lemma 3.2. *If $M \approx N$, then $M =_\beta N$.*

The other way around, we can map \mathcal{T} into \mathcal{T}_∞ by adding pseudo-terms as labels to the free variables, and to the free variables in those labels etc., which should end in a closed pseudo-term. This should of course be done *consistently*, i.e. the same free variable should receive exactly the same labelling throughout a pseudo term.

Definition 3.3. *Given $M \in \mathcal{T}$ with $\text{FV}(M) = \{x_1, \dots, x_n\}$ and $N_1, \dots, N_n \in \mathcal{T}_\infty$, we say that $M[x_1 := x_1^{N_1}, \dots, x_n := x_n^{N_n}]$ is a consistent labelling of M .*

We have the following immediate corollary of this definition.

Lemma 3.4. *If $M =_\beta N$ and $M^\infty N^\infty$ is a consistent labelling of $M N$, then $M^\infty \approx N^\infty$.*

In a PTS, the context takes care that one can only abstract over a variable if nothing else depends on that variable. In the rules, this is formalized by requiring that the $x : A$ that one abstracts over in the Π or λ rule must be the last declaration in the context. This ensures that x does not occur free in any of the other types in the context. In \mathcal{T}_∞ we don’t have contexts, so we have to replace

this by another side condition on the rules. We would like to have the Π rule as follows:

$$\frac{A : s_1 \quad B : s_2}{\Pi x_j : A. B[x_i^A := x_j] : s_3} \quad (s_1, s_2, s_3) \in \mathcal{R}$$

but that is wrong, because then we would be able to form (in PTS terminology)

$$\frac{\dots \vdash A : * \quad A : *, P : A \rightarrow *, Q : \Pi x : A. P x \rightarrow *, a : A, h : P a \vdash Q a h : *}{A : *, P : A \rightarrow *, Q : \Pi x : A. P x \rightarrow *, h : P a \vdash \Pi x_j : A. Q x_j h : *}$$

This is not correct, because h is not of type $P x_j$ in the conclusion. In Γ_∞ , this derivation would read (adding some brackets for readability):

$$\frac{A^* : * \quad Q^{\Pi x_1 : A^*. (P^{A^* \rightarrow *} x_1) \rightarrow *} a^{A^*} h^{P^{A^* \rightarrow *} a^{A^*}} : *}{\Pi x_2 : A. Q^{\Pi x_1 : A. (P^{A^* \rightarrow *} x_1) \rightarrow *} x_2 h^{P^{A^* \rightarrow *} a^{A^*}} : *}$$

which would be well-formed according to the Π -rule above, but clearly not what we want.

Definition 3.5. *Given $M \in \mathcal{T}_\infty$, we define the hereditary free variables in M , denoted $\text{hfv}(M)$, as follows:*

$$\begin{aligned} \text{hfv}(s) &= \text{hfv}(x_i) = \emptyset \\ \text{hfv}(x_i^t) &= \{x_i^t\} \cup \text{hfv}(t) \\ \text{hfv}(t q) &= \text{hfv}(t) \cup \text{hfv}(q) \\ \text{hfv}(\lambda x_i : t. q) &= \text{hfv}(\Pi x_i : t. q) = \text{hfv}(t) \cup \text{hfv}(q) \end{aligned}$$

So, where \approx basically ignores the structure of the type labels of the free variables, we now take them seriously, by collecting the (hereditary) free variables in the type labels as well.

We put as a side condition in the Π -rule that x_i^A should not occur free in any of the *types of the free variables in B* , and similarly for the λ rule. We give an explicit definition of this notion.

Definition 3.6. *Given $M \in \mathcal{T}_\infty$, we define the hereditary free variables of the types of the free variables in M , denoted $\text{hfvT}(M)$, as follows:*

$$\begin{aligned} \text{hfvT}(s) &= \text{hfvT}(x_i) = \emptyset \\ \text{hfvT}(x_i^t) &= \text{hfvT}(t) \\ \text{hfvT}(t q) &= \text{hfvT}(t) \cup \text{hfvT}(q) \\ \text{hfvT}(\lambda x_i : t. q) &= \text{hfvT}(\Pi x_i : t. q) = \text{hfvT}(t) \cup \text{hfvT}(q) \end{aligned}$$

So, for example $\text{hfvT}(h^{P^{A^* \rightarrow *} a^{A^*}}) = \{P^{A^* \rightarrow *}, a^{A^*}, A^*\}$. An easy corollary of the definition is that

$$\text{hfvT}(M) \subseteq \text{hfv}(M)$$

We now give the derivation rules of the system.

Definition 3.7. *The derivation rules of Γ_∞ are*

(sort) $\frac{}{s_1 : s_2}$	$if (s_1, s_2) \in \mathcal{A}$
(var) $\frac{A : s}{x_i^A : A}$	
(Π) $\frac{A : s_1 \quad B : s_2}{\Pi x_j : A. B[x_i^A := x_j] : s_3}$	$if (s_1, s_2, s_3) \in \mathcal{R} \text{ and } x_i^A \notin \text{hfvT}(B)$
(λ) $\frac{M : B \quad \Pi x : A. B[x_i^A := x_j] : s}{\lambda x_j : A. M[x_i^A := x_j] : \Pi x_j : A. B[x_i^A := x_j]}$	$if x_i^A \notin \text{hfvT}(M, B)$
(app) $\frac{M : \Pi x_i : A. B \quad N : A}{MN : B[x_i := N]}$	
(conv) $\frac{M : A \quad B : s}{M : B}$	$A \approx B$

The Π and λ rules have the side condition that x_j should not occur at all – not even in a binding – in B or M .

The side condition on the Π and λ rules is no restriction as you can go to an α -equivalent version of the term afterwards.

4 The correspondence theorems

We now prove that a PTS and its Γ_∞ -variant correspond to each other. For a Γ_∞ judgment $M : A$ we generate a context Γ such that $\Gamma \vdash M : A$ is typable in the PTS. As all free variables in Γ_∞ are type-annotated and we want to use the same names for the free variables in the PTS, the context Γ will consist of declarations of the form $x^B : B$. From the PTS point of view the superscript B is just a string that identifies the variable. However, choosing the free variable names in such a way will be very convenient in the correspondence proof.

Definition 4.1. *A type annotated context in a PTS is a context of the form*

$$x_1^{B_1} : B_1, \dots, x_n^{B_n} : B_n$$

where we moreover assume that all bound variables in the B_i are not labelled.

In a type annotated context, the free variables are labelled by their own type. Therefore, if M is typable in an annotated context then in M as well, all free variables are labelled by their own type. We straightforwardly extend the notion of hfvT to terms in annotated contexts.

Lemma 4.2. *If Γ and Δ are type annotated contexts, then $\Gamma \parallel \Delta$.*

Proof. If $\Gamma = x_1^{A_1} : A_1, \dots, x_n^{A_n} : A_n$ and $\Delta = y_1^{B_1} : B_1, \dots, y_m^{B_m} : B_m$, then for all i and j , $x_i^{A_i} \neq y_j^{B_j}$, so $\Gamma \parallel \Delta$.

Lemma 4.3. *If Γ is a type annotated context with $x^A \in \text{dom}(\Gamma)$, $\Gamma \vdash M : B$ and $x^A \notin \text{hfvT}(M, B)$, then*

$$\exists \Delta \subseteq \Gamma(\Delta, x^A : A \vdash M : B)$$

Proof. Write $\Gamma = \Gamma_1, x^A : A, \Gamma_2$, and suppose $y^C \in \text{dom}(\Gamma_2)$ with $x^A \in \text{FV}(C)$. If $y^C \in \text{FV}(M, B)$, then $x^A \in \text{hfvT}(M, B)$, contradiction. So $y^C \notin \text{FV}(M, B)$. This means that all declarations $y^C : C$ in Γ_2 for which $x^A \in \text{FV}(C)$ can be removed by Strengthening (Proposition 2.3), starting from the rightmost declaration in Γ_2 . We end up with a judgment

$$\Gamma_1, x^A : A, \Gamma'_2 \vdash M : B$$

with $\Gamma'_2 \subset \Gamma_2$ and $x^A \notin \text{type}(\Gamma'_2)$. Using Permutation (Proposition 2.3), we conclude that $\Gamma_1, \Gamma'_2, x^A : A \vdash M : B$ and we take Γ_1, Γ'_2 for Δ .

Lemma 4.4. *Let $M : A$ be a derivable Γ_∞ judgment. Then there is a judgment $\Gamma \vdash M : A$, derivable in the associated PTS, such that Γ contains all free variables of the form x_i^A occurring in M, A .*

Proof. By induction on the derivation of $M : A$ we define the type annotated context $\Gamma(M, A)$ and we prove that $\Gamma(M, A) \vdash M : A$ in the PTS (using Lemma 4.2).

- (var) By induction, $\Gamma(A, s) \vdash A : s$. So $\Gamma(A, s), x_i^A : A \vdash x_i^A : A$.
- (conv) By induction, $\Gamma(M, A) \vdash M : A$ and $\Gamma(A', s) \vdash A' : s$, and we also know that $A =_\beta A'$. So $\Gamma(M, A) \times \Gamma(A', s) \vdash M : A'$ by Thinning and the (conv) rule.
- (app) By induction, $\Gamma(F, \Pi x_i : A.B) \vdash F : \Pi x_i : A.B$ and $\Gamma(M, A) \vdash M : A$. So $\Gamma(F, \Pi x_i : A.B) \times \Gamma(M, A) \vdash F M : B[x_i := M]$ by Thinning and the (app) rule.
- (II) By induction, $\Gamma(A, s_1) \vdash A : s_1$ and $\Gamma(B, s_2) \vdash B : s_2$.
 If $x_i^A \notin \Gamma(B, s_2)$, then $\Gamma(A, s_1) \times \Gamma(B, s_2), x_i^A : A_i \vdash B : s_2$, so by Thinning and the (II) rule we have $\Gamma(A, s_1) \times \Gamma(B, s_2) \vdash \Pi x_j : A.B[x_i^A := x_j] : s_3$.
 If $x_i^A \in \Gamma(B, s_2)$, then $\Delta, x_i^A : A \vdash B : s_2$ for some $\Delta \subset \Gamma(B, s_2)$. So $\Gamma(A, s_1) \times \Delta \vdash \Pi x_j : A.B[x_i^A := x_j] : s_3$ by Thinning and the (II) rule.
- (λ) By induction, $\Gamma(M, B) \vdash M : B$ and $\Gamma(\Pi x_j : A.B[x_i^A := x_j], s) \vdash \Pi x_j : A.B[x_i^A := x_j] : s$.
 If $x_i^A \notin \Gamma(M, B)$, then $\Gamma(A, s_1) \times \Gamma(M, B), x_i^A : A_i \vdash M : B$ and so $\Gamma(\Pi x_j : A.B[x_i^A := x_j], s) \times \Gamma(M, B) \vdash \lambda x_j : A.M[x_i^A := x_j] : \Pi x_j : A.B[x_i^A := x_j]$ by Thinning and the (λ) rule.
 If $x_i^A \in \Gamma(M, B)$, then $\Delta, x_i^A : A \vdash M : B$ for some $\Delta \subset \Gamma(M, B)$. So $\Gamma(\Pi x_j : A.B[x_i^A := x_j], s) \times \Delta \vdash \lambda x_j : A.M[x_i^A := x_j] : \Pi x_j : A.B[x_i^A := x_j]$ by Thinning and the (λ) rule.

Theorem 4.5. *Let*

$$M : A$$

be a derivable Γ_∞ judgment. Take all variables of the form x_i^A occurring in it, and put them in any order $x_{i_1}^{A_1}, \dots, x_{i_n}^{A_n}$ such that if $x_{i_k}^{A_k}$ occurs in A_l then $k < l$. (I.e., topologically sort them in some order.) Then

$$x_{i_1}^{A_1} : A_1, \dots, x_{i_n}^{A_n} : A_n \vdash M : A$$

is derivable in the PTS.

Proof. Corollary of the Lemma, using Strengthening.

We now prove the other direction: if a judgment $\Gamma \vdash M : A$ is derivable in a PTS, then $M : A$ is derivable in the associated Γ_∞ system. To do this, we have to add ‘type labels’ to the free variables in M and A . These depend only on Γ , so we define this labelled version of M , M_Γ , first.

Definition 4.6. *Given a context Γ and a pseudo-term M , we define $M_\Gamma \in \mathcal{T}_\infty$ as follows.*

$$\begin{aligned} s_\Gamma &= s \\ x_\Gamma &= x \text{ if } x \notin \text{dom}(\Gamma) \\ x_\Gamma &= x^{A_\Gamma} \text{ if } x:A \in \Gamma \\ (tq)_\Gamma &= t_\Gamma q_\Gamma \\ (\lambda x:t.q)_\Gamma &= \lambda x:t_\Gamma.q_\Gamma \\ (\Pi x:t.q)_\Gamma &= \Pi x:t_\Gamma.q_\Gamma \end{aligned}$$

We remark that, if $x \notin \text{FV}(\text{type}(\Gamma))$, then $x \notin \text{hfvT}(M_\Gamma)$. We have a substitution Lemma of a restricted form:

Lemma 4.7. *If $x \notin \text{dom}(\Gamma)$, then*

$$(M[x := N])_\Gamma = M_\Gamma[x := N_\Gamma].$$

Proof. The only relevant case to consider is if $M \equiv x$. Then $(x[x := N])_\Gamma = N_\Gamma = x[x := N_\Gamma] = x_\Gamma[x := N_\Gamma]$.

If $\Gamma \vdash M : A$, then M_Γ and A_Γ are well-defined (and moreover consistently labelled) and $M_\Gamma : A_\Gamma$ in Γ_∞ , as the next Lemma states.

Lemma 4.8. *If $\Gamma \vdash M : A$ in the PTS, then $M_\Gamma : A_\Gamma$.*

Proof. By induction on the derivation of $\Gamma \vdash M : A$:

- (var) By induction, $A_\Gamma : s$, so $x_\Gamma = x^{A_\Gamma} : A_\Gamma$.
- (conv) By induction, $M_\Gamma : A_\Gamma$ and $A'_\Gamma : s$. Moreover $A_\Gamma \approx A'_\Gamma$, so $M_\Gamma : A'_\Gamma$.
- (app) By induction, $F_\Gamma : \Pi x_i : A_\Gamma.B_\Gamma$ and $M_\Gamma : A_\Gamma$, so $(FM)_\Gamma = F_\Gamma M_\Gamma : B_\Gamma[x_i := M_\Gamma] = (B[x_i := M])_\Gamma$.

- (II) By induction, $A_\Gamma : s_1$ and $B_{\Gamma, x:A} : s_2$. We also know that $x \notin \text{FV}(\text{type}(\Gamma, x:A))$ and therefore $x \notin \text{hfvT}(B_{\Gamma, x:A})$, so we can apply the (II) rule in Γ_∞ . Because $x, x_j \notin \text{dom}(\Gamma)$, we find that $(\Pi x_j:A. B[x := x_j])_\Gamma = \Pi x_j:A_\Gamma. B_\Gamma[x := x_j] : s_3$.
- (λ) By induction, $(\Pi x_j:A. B[x := x_j])_\Gamma : s$ and $M_{\Gamma, x:A} : B_{\Gamma, x:A}$. We also know that $x \notin \text{type}(\Gamma, x:A)$ and therefore $x \notin \text{hfvT}(M_{\Gamma, x:A}, B_{\Gamma, x:A})$, so we can apply the (λ) rule. Because $x, x_j \notin \text{dom}(\Gamma)$, we find that $(\lambda x_j:A. M[x := x_j])_\Gamma = \lambda x_j:A_\Gamma. M_\Gamma[x := x_j] : \Pi x_j:A. B[x := x_j]_\Gamma = \Pi x_j:A_\Gamma. B_\Gamma[x := x_j]$.

Theorem 4.9. *Let*

$$x_1^{A_1} : A_1, \dots, x_n^{A_n} : A_n \vdash M : A$$

be a derivable judgment of the PTS. (We can always rename the declared variables in the context to have these particular names, by Thinning, so this is not a restriction on the judgments that we are considering.) Now if the bound variables in A , M , and in all of the A_i all are of the form x_j , then

$$M : A$$

is derivable in Γ_∞ .

Proof. Corollary of the Lemma, since we have $(M)_\Gamma = M$ and $(A)_\Gamma = A$.

5 Conclusion

5.1 Future work

There are two obvious continuations of this work:

1. The first is to investigate to what extend other type theories than the PTSs admit a Γ_∞ presentation.
2. The second is to see how well the approach presented here can be used as a basis of an LCF-style kernel for type theory.

As far as the first point is concerned: we expect most type theories to have a Γ_∞ -variant without complications. Most importantly it will be to investigate how our approach needs to be changed to support *type theories with definitions* [10]. In fact, in an implementation of a real system, the definitions for defined constants will be a more significant part of the contexts Γ that we are getting rid of, than the free variables.

The second point we are currently already investigating, by developing an ocaml implementation for the PTS λP (a system that corresponds to the logical framework LF) along the lines of this paper. The main questions about this implementation will be how expensive, computationally, the two following operations have to be:

- The substitutions $[x_i^A := x_j]$ that occur in the λ and Π rules.
- The check of the side-condition $x_i^A \notin \text{hfvT}(M, B)$ in the λ and Π rules.

The first check in some sense is ‘local’, because it does not look inside definitions of constants. To make the second check reasonably efficient will be harder. It might be the case that we will need to go to *three* kind of variables, by distinguishing x_i^A variables that are allowed to be substituted with bound variables but which are not allowed to remain when a constant is defined from x_i^A variables that are considered to be ‘axiomatic constants’ of the system.

One important aspect of an implementation like this is that it is essential that the implementation language can use pointer equality to efficiently determine equality of terms. This motivated our choice for ocaml, which is one of these languages. Although in ocaml the comparison ‘=’ does not have this feature (because floating points NaNs are not taken to be equal to themselves, which causes the system to never look at pointer equality when evaluating this relation), the comparison ‘`fun x y -> Pervasives.compare x y = 0`’ does.

5.2 Discussion

We would like to stress that we do not claim that the presentation of type theory that we propose here is better than the standard one. It just is different. In fact, we think that for most purposes the standard one probably is the more convenient one. However, we would still like to have the approach from this paper as an alternate possibility in the literature.

Acknowledgments. Thanks to Jean-Christophe Filliâtre for details about the architecture of the Coq kernel.

References

1. Henk Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2: Background: Computational Structures*, pages 117–309. OUP, 1992.
2. Henk Barendregt and Herman Geuvers. Proof Assistants using Dependent Type Systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.
3. N.G. de Bruijn. Wees contextbewust in WOT. *Euclides*, 55:7–12, 1979.
4. Herman Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, The Netherlands, 1993.
5. Joachim Lambek and Phil J. Scott. *Introduction to Higher-Order Categorical Logic*. Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1988.
6. Karel Lambert. Existential import revisited. *Notre Dame J. Formal Logic*, 4(4):288–292, 1963.
7. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23, 1999.
8. Mike Gordon and Robin Milner and Chris Wadsworth. *Edinburgh LCF: A mechanised logic of computation*. Number 78 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

9. Randy Pollack. Closure under alpha-conversion. In *TYPES '93: Proceedings of the international workshop on Types for proofs and programs*, volume 806 of *Lecture Notes in Computer Science*, pages 313–332. Springer-Verlag, 1994.
10. Paula Severi and Erik Poll. Pure type systems with definitions. In *Logical Foundations of Computer Science '94*, volume 813 of *Lecture Notes in Computer Science*, pages 316–328. Springer-Verlag, 1994.
11. Jeffrey A. Vaughan. A review of three techniques for formally representing variable binding, 2006. University of Pennsylvania CIS Technical Report Number MS-CIS-06-19. <http://www.seas.upenn.edu/~vaughan2/docs/wpe-ii.pdf>.