Pure Type Systems without Explicit Contexts

Herman Geuvers
Radboud University Nijmegen
Technical University Eindhoven
The Netherlands

Robbert Krebbers   James McKinna   Freek Wiedijk
Radboud University Nijmegen
Institute for Computing and Information Sciences
Heyendaalseweg 135, 6525 AJ Nijmegen
The Netherlands

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

Specifically we give the rules of the 'Pure Type System' class of type theories in this style. We prove that the typing judgments of these systems correspond in a natural way with those of Pure Type Systems as traditionally formulated. 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 gives us is the need to be aware of the *context* in which one works. This was already stressed by de Bruijn in his 1979 paper *Wees contextbewust in WOT* [6], Dutch for "Be context aware in the mathematical vernacular". In type theory a term always is considered with respect to a context Γ, which gives the types of the variables occurring free in the term. This is also apparent in the shape $\Gamma \vdash M : A$ of the judgments of type theory, where the context Γ is made explicit. Thus a 'bound' variable is bound *locally* in a term, while a 'free' variable actually is *globally* bound, namely by the context.

In customary presentations of first order predicate logic [16, 22, for example], 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 called *free* logic [12]. In traditional treatments the formula $(\forall x.\, P(x)) \to (\exists x.\, P(x))$ is usually provable. For instance a natural deduction proof of this formula would look like:

LFMTP 2010
EPTCS ??, 20??, pp. 1–15, doi:10.4204/EPTCS.??.??

$$\dfrac{\dfrac{\dfrac{[\forall x.\, P(x)]}{P(y)}\,\forall E}{\exists x.\, P(x)}\,\exists I}{(\forall x.\, P(x)) \to (\exists x.\, P(x))}\,{\to}I$$

This proof uses the free variable y. If one cannot use any other variables than those introduced by earlier rules, then this proof fails. Indeed, the type corresponding to the formula is not inhabited: there is no term M such that the following judgment is derivable:

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

because we cannot avoid the case in which the domain D is empty, where the formula is false.

Now there are two things one can do to bridge this gap 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 some infinite supply.

Although the first option is interesting too, especially in categorical treatments of logic [11, 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.

In those type theories actually implemented in interactive theorem provers, the context always consists of a part holding global *definitions* and parameters and a part holding the *free variables* in the term (as in [19, 21, for example]). For simplicity of exposition, and for the sake of proving an exact correspondence between a standard presentation of type theory and the variant we propose, in this paper we consider only the second part of such contexts. We believe, however, that the other part can be treated in exactly the same way.

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 [9]. In the original form of such a system there is an abstract data type called `term`, whose elements can only be created by a small number of functions exported from the type-checking kernel. Elements of this datatype always correspond to type-correct terms, and those terms can contain free variables.

A system using this approach has a kernel interface containing a function:

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

When this function is called, the kernel of the system makes sure that the types of the arguments are compatible, i.e., that the result is again a type-correct term.

This is how the HOL family of theorem provers is implemented. These systems have a logical foundation that is based on a typed lambda calculus. However, in these systems the free variables in the terms are not recorded in a context of variables. The only context in these systems is the context of definitions, which is kept track of in a stateful variable. Definitions are never allowed to be removed from this context, as that would compromise the safety of the kernel. Hence, these systems are stateful, although they can be made functional using a variant of the approach presented here [24].

There are two classes of systems that can be said to implement a *type theory*, a typed lambda calculus:

- The simpler type theories, in which no dependent types are allowed. They often are a form of simple type theory with some enhancements, such as some form of polymorphism or type classes. These include the systems from the HOL family: HOL4, HOL Light, ProofPower and Isabelle. These systems can be, and are, implemented following the LCF architecture just outlined. In these systems variables come from an infinite 'sea' of free variables, and in the logical theory there is *no* context keeping track of the variables.
- More advanced type theories, often called type theory with *dependent types*. These come from the Dutch AUTOMATH systems, the Swedish tradition of Martin-Löf type theory, the French tradition of the Calculus of Constructions and variations on the Edinburgh Logical Framework. Their implementations include Coq, NuPRL, Twelf, Agda, Lego, Plastic and Epigram. In the logical theory of these systems there *is* a context keeping track of the variables.

For this second style of type theory the pure LCF 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.

For this reason actual type-theoretical proof assistants have a kernel with a different kind of interface. In such a kernel there is no abstract datatype of *terms* (there just is a—non-abstract—type of *pseudo-terms*). Instead there is an abstract type of well-typed *contexts*, which we call `context` here. (There is also a type `pseudocontext`, but that is irrelevant here.) The interface then looks like:

```
mkApp : pseudoterm * pseudoterm -> pseudoterm
add_constant : string * pseudoterm -> context -> context
```

where `mkApp` is just the constructor of the data type of pseudoterms, whereas `add_constant` is a function that does the type-checking: a pseudoterm will only be added to the context after it has been type-checked. (These are the actual names of the functions in the kernel of the Coq system. The types of those functions in Coq are essentially what is presented here. The type `pseudoterm` is called `constr` in Coq, while the type `context` is called `safe_environment`.) The system also has a global variable

```
global_env : context ref
```

corresponding to the *state* of the system in which the user works. It is not part of the kernel (and in fact is changed back by an undo operation), but as there are no interesting operations combining two different `contexts`, only one global `context` is ever relevant, the one given by the contents of this variable.

Although the architecture with contexts 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 a possible 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 we present here will lead to such a type-checking architecture.

1.2 Approach

The approach 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*. Reduction will never happen inside these As. Also, x_i^A and x_i^B will be *different* variables, even when A and B are convertible, or even if they are α-equivalent. Note that the (free) variables in A themselves will also be of shape y_j^B: this means that the variables themselves, as well as the terms, have a recursive tree-like structure.

For example, a variable corresponding to the successor function on natural numbers looks like:

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

If we use numbered names for the variables, this might become:

$$x_0^{x_0^* \to x_0^*}$$

So the "small price" alluded to above is that a free variable x_i^A in a well-typed term carries with it the (well-founded) history of how it comes to be well-typed; that is, the label A witnesses the validity of the context extension $\Gamma, x_i : A$.

Now our systems will have judgments $A : B$, which should be interpreted as $\Gamma_\infty \vdash A : B$. For this reason we call the general approach to type theory introduced 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 we give here uses named variables for them as well.

Although we expect many type theories to have a Γ_∞-variant, here we only consider the class of type theories called Pure Type Systems [2] (PTSs). That way we keep everything concrete, and it allows us to prove a precise correspondence between PTSs in the traditional style and our version in Γ_∞ style.

One should note that in Γ_∞ any type will be inhabited, simply because there are free variables of every type: in particular, just as in traditional treatments of logic, all domains are assumed to be inhabited. This is in essence the same as in the version with contexts, except that such variables are there explicitly recorded in the context.

1.3 Related Work

To our surprise, we found little published work investigating such an approach to *dependent* type theory. In Church's original formulation of *simple* type theory [5], variables, both free *and* bound, are annotated with their types, writing for example $\lambda x^\alpha . f^{\alpha\to\alpha} x^\alpha$. (whereas in our formulation we would write $\lambda x{:}\alpha^* . f^{\alpha^*\to\alpha^*} x$.) Girard adopted 'Church-style' in the account of System F in his thesis [8]. In neither system do *term* variables occur in types, while *type* variables are not regulated by an explicit context. In these non-dependent type theories, contexts are not strictly needed, because one can define the different syntactic classes — types, terms — in stages. One can regard our approach as extending that of Church to dependent types, but optimised to avoid the need to consider substitution in labels on *bound* variables which otherwise might arise in the application rule.

Conor McBride (private communication) observed that Pollack's LEGO implementation already supported the Γ_∞ idea, and this idea was then used in his OLEG extensions, and subsequently in the architecture of EPIGRAM 1. 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 [15]. The motivation there was to avoid capture during substitution while keeping a close correspondence with the informal presentation of PTSs with named variables, rather than how to give a 'Γ_∞' presentation, as here. Various approaches to representing binding are discussed in [23], which considers named free variables and de Bruijn index bound variables one of the best options for mechanisation. Indeed, in ongoing work [10] the second author has formalised one half of the correspondence proved in Section 4 in such a style. We expand on the niceties of this formalisation in Section 5.

Elsewhere, Pollack considered presentations of type theory separating the typing judgment from that for context well-formedness [17], although judgments are still 'in context'. This allows a subtle range of issues to be explored, especially regarding closure under α-conversion, here treated only informally.

Most significantly, but starting from a rather different point, Sacerdoti Coen considered the problem of proof-checking in the setting of a distributed library, and hence the problem of how to *reconstruct* a context in which a given term may be successfully type-checked [19]. This work (elaborated in [18], and forming the basis of the Matita system) goes beyond the standard PTS setting considered in this paper. It identifies a subtle problem which arises when attempting to merge contexts (including definitions) in the presence of global constraints (such as universe levels).

Added in proof. Between acceptance of the final version of this paper and this final version, our attention was drawn to the recent work of Matthias Boespflug [4], which itself references an earlier (2009) account of our ideas. By contrast with our presentation, which is purely first-order, Boespflug uses higher-order abstract syntax (HOAS), with a view to implementation.

1.4 Contribution

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

We validate our approach by proving two theorems, 13 and 19 below, establishing a straightforward correspondence between the standard presentation and the variant presented 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) generalize many existing type systems and thus the class of PTSs contains various well-known systems, like systems F and Fω, dependent type theory λP and the Calculus of Constructions.

Definition 1. *For $\mathscr{S}$ a set (the set of* sorts*), $\mathscr{A} \subseteq \mathscr{S} \times \mathscr{S}$ (the set of* axioms*) and $\mathscr{R} \subseteq \mathscr{S} \times \mathscr{S} \times \mathscr{S}$ (the set of* rules*), the* Pure Type System *$\lambda(\mathscr{S},\mathscr{A},\mathscr{R})$ is the typed λ-calculus with inference rules as in Figure 1. In the rules, the expressions M,N,A,B,C are taken from the set of* pseudo-terms *$\mathscr{T}$ defined by*

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

with $\mathscr{V}$ a set of variables, and the Γ taken from the set of pseudo-contexts

$$x_1 : A_1,\ldots,x_n : A_n \quad (x_i \in \mathscr{V}, A_i \in \mathscr{T}, 1 \le i \le n)$$

with the x_i all distinct. (We leave the choice of variable names *unspecified at this point, as this does not matter as long as $\mathscr{V}$ is countably infinite, but below we will take a specific choice of names.)*

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

Figure 1: Typing rules for PTSs

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 [2, 3, 7] for details and explanation. Here we just give the results that we use in the rest of the paper to prove the equivalence between a PTS and its Γ_∞-variant.

Definition 2. *The pseudo-term A is called* well-typed *if a pseudo-context Γ and pseudo-term B exist such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$ is derivable. A pseudo-context Γ is* well-formed *if pseudo-terms A and B exist such that $\Gamma \vdash A : B$ is derivable; a* context *is a well-formed pseudo-context. The set of variables declared in pseudo-context Γ is called the* domain *of Γ,* $\mathrm{dom}(\Gamma)$. *For $x \in \mathrm{dom}(\Gamma)$, let $\mathrm{type}_\Gamma(x)$ denote the 'type' assigned to x in Γ: if $x : A \in \Gamma$, then $\mathrm{type}_\Gamma(x) = A$. The expression $\mathrm{type}(\Gamma)$ denotes the set of such 'types' occurring in Γ. The set of well-typed terms of $\lambda(\mathscr{S}, \mathscr{A}, \mathscr{R})$ is denoted by* $\mathrm{Term}(\lambda(\mathscr{S}, \mathscr{A}, \mathscr{R}))$.

We adopt the usual notions of bound and free variable, α-conversion ($\equiv$), substitution ($B[x := N]$, used in the rule (app)), β-reduction ($\rightarrow_\beta$) and β-equality ($=_\beta$, used in the rule (conv)) on pseudo-terms.

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

Proposition 3.

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

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

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

In proving the equivalence between a PTS and its Γ_∞-variant, we need to merge two contexts to create a new one. Therefore we introduce the following:

Definition 4. *Let* Γ *and* Δ *be two pseudo-contexts. We say* Γ *and* Δ *are* compatible*, notation* $\Gamma \mid\mid\mid \Delta$*, if*

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

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

Note the strong requirement in $\Gamma \mid\mid\mid \Delta$ that the types of x in Γ and Δ should be α-equal, and not just β-convertible.

Lemma 5. *If* Γ *and* Δ *are contexts and* $\Gamma \mid\mid\mid \Delta$*, then* $\Gamma \ltimes \Delta$ *is well-formed.*

Proof. We write $x_1 : B_1, \ldots, x_n : B_n$ for Δ. $\Gamma \ltimes \Delta$ is the pseudo-context $\Gamma, (\Delta \setminus \Gamma)$. As Γ is well-formed, we only have to consider the part $\Delta \setminus \Gamma = x_{i_1} : B_{i_1}, \ldots, x_{i_n} : B_{i_n}$. But $x_1 : B_1, \ldots, x_{i_1 - 1} : B_{i_1 - 1} \subseteq \Gamma$, 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 $x_{i_2} : B_{i_2}, \ldots, x_{i_n} : B_{i_n}$, so we conclude that $\Gamma, (\Delta \setminus \Gamma)$ is well-formed. □

In our system Γ_∞, the free variables will have special names, as they are labelled by their types. Of course, consistently renaming the free variables in a judgment $\Gamma \vdash M : A$ does not change its meaning, as the free variables in M and A are actually bound in Γ. For clarity, we introduce this notion explicitly.

Definition 6. *The judgment* $\Gamma \vdash M : A$ *is* α*-equivalent to* $\Gamma' \vdash M' : A'$ *in case one can be obtained from the other by renaming bound variables, where we consider the free variables in* M *and* A *to be bound by their declaration in* Γ*.*

3 Pure Type Systems in the Γ_∞ style

We now make the set of variables $\mathscr{V}$ explicit. We have two kinds: variables $\dot{x}$ with a dot on top, intended to be bound by λ and Π binders; and variables x^A tagged with a pseudo-term A, intended to be bound in the context. This means we take $\mathscr{V}$ in Definition 1 as follows, where $\mathscr{X}$ supplies the *names* of variables:

$$\begin{array}{rcl} \mathscr{V} & ::= & \dot{\mathscr{X}} \mid \mathscr{X}^{\mathscr{T}} \\ \mathscr{X} & ::= & x \mid y \mid z \mid \ldots \mid x_0 \mid x_1 \mid x_2 \mid \ldots \end{array}$$

Clearly the rules for $\mathscr{T}$ and $\mathscr{V}$ are mutually recursive.

Note that although the variables are *intended* to be used in a certain way (made precise in Definition 10 below), in the PTSs as defined above both kinds of $\mathscr{V}$ can be used for all purposes. In particular $\dot{x}$ can be put in a context, x^A can be bound, and the label A of x^A need not correspond to the type of x^A.

Note also that the definition of substitution, and hence the relation of β-equality, is agnostic about the structure of the annotations of the variables. (The definition of $=_\beta$ in Section 2 takes $\mathscr{V}$ just as a set). This means that although

$$(\lambda \dot{A} : *.\dot{A})\, B^* =_\beta B^*$$

we have that

$$x^{(\lambda \dot{A}:*.\dot{A})B^*} \neq_\beta x^{B^*}$$

We will now define the rules of Γ_∞. To do this we first have to introduce the notion of *hereditarily free variables of the types of the free variables* in a term. We first motivate the need for this notion.

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 formalised by requiring that the $x : A$ abstracted 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 Γ_∞ we do not have contexts, so we have to replace this by another side condition on the rules. We would like to have a Π rule as follows:

$$\frac{A : s_1 \qquad B : s_2}{\Pi\dot{y}{:}A.\,B[x^A := \dot{y}] : s_3}\ (s_1, s_2, s_3) \in \mathscr{R}$$

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

$$\frac{\ldots \vdash A : * \qquad A : *, P : A \to *, Q : \Pi x{:}A.P\,x \to *, a : A, h : P\,a \vdash Q\,a\,h : *}{A : *, P : A \to *, Q : \Pi x{:}A.P\,x \to *, h : P\,a \vdash \Pi y : A.Q\,y\,h : *}$$

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

$$\frac{A^* : * \qquad Q^{\Pi\dot{x}:A^*.(P^{A^* \to *}\,\dot{x}) \to *}\,a^{A^*}\,h^{P^{A^* \to *}\,a^{A^*}} : *}{\Pi\dot{y} : A^*.Q^{\Pi\dot{x}:A^*.(P^{A^* \to *}\,\dot{x}) \to *}\,\dot{y}\,h^{P^{A^* \to *}\,a^{A^*}} : *}$$

which would be derivable according to the Π-rule above, but clearly undesirable.

Definition 7. *Given $M \in \mathscr{T}$, we define the* hereditarily free variables in *M, denoted* $\mathrm{hfv}(M)$*, as follows:*

$$\begin{array}{rcl} \mathrm{hfv}(s) = \mathrm{hfv}(\dot{x}) & = & \emptyset \\ \mathrm{hfv}(x^A) & = & \{x^A\} \cup \mathrm{hfv}(A) \\ \mathrm{hfv}(F\,N) & = & \mathrm{hfv}(F) \cup \mathrm{hfv}(N) \\ \mathrm{hfv}(\lambda\dot{x}{:}A.N) = \mathrm{hfv}(\Pi\dot{x}{:}A.N) & = & \mathrm{hfv}(A) \cup \mathrm{hfv}(N) \end{array}$$

So, where $=_\beta$ basically ignores the structure of the type labels of free variables, we now take them seriously, collecting the variables (hereditarily) free in the type labels as well.

We put as a side condition in the Π-rule that x^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 8. *Given $M \in \mathscr{T}$, we define the* hereditarily free variables of the types of the free variables in *M, denoted* $\mathrm{hfvT}(M)$*, as follows:*

$$\begin{array}{rcl} \mathrm{hfvT}(s) = \mathrm{hfvT}(\dot{x}) & = & \emptyset \\ \mathrm{hfvT}(x^A) & = & \mathrm{hfv}(A) \\ \mathrm{hfvT}(F\,N) & = & \mathrm{hfvT}(F) \cup \mathrm{hfvT}(N) \\ \mathrm{hfvT}(\lambda\dot{x}{:}A.N) = \mathrm{hfvT}(\Pi\dot{x}{:}A.N) & = & \mathrm{hfvT}(A) \cup \mathrm{hfvT}(N) \end{array}$$

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

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

We now give the derivation rules of the system.

Definition 9. *The derivation rules of Γ_∞ are given by the inference rules in Figure 2. The Π and λ rules have the side condition that $\dot{x}$ should not be* captured *in B or M when doing the substitution. That is, $\dot{x}$ should not be bound by a binder under which y^A occurs.*

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.

$$
\begin{array}{lll}
\text{(sort)} & \dfrac{}{s_1 : s_2} & \text{if } (s_1, s_2) \in \mathscr{A} \\[2ex]
\text{(var)} & \dfrac{A : s}{x^A : A} & \\[2ex]
(\Pi) & \dfrac{A : s_1 \quad B : s_2}{\Pi \dot{x}{:}A.B[y^A := \dot{x}] : s_3} & \text{if } (s_1, s_2, s_3) \in \mathscr{R} \text{ and } y^A \notin \mathrm{hfvT}(B) \\[2ex]
(\lambda) & \dfrac{M : B \quad \Pi \dot{x}{:}A.B[y^A := \dot{x}] : s}{\lambda \dot{x}{:}A.M[y^A := \dot{x}] : \Pi \dot{x}{:}A.B[y^A := \dot{x}]} & \text{if } y^A \notin \mathrm{hfvT}(M) \cup \mathrm{hfvT}(B) \\[2ex]
\text{(app)} & \dfrac{M : \Pi \dot{x}{:}A.B \quad N : A}{MN : B[\dot{x} := N]} & \\[2ex]
\text{(conv)} & \dfrac{M : A \quad B : s}{M : B} & A =_\beta B
\end{array}
$$

Figure 2: Typing rules for Γ_∞

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 PTS-derivable. Conversely, if $\Gamma \vdash M : A$ is PTS-derivable, we always have an α-equivalent judgment (see Definition 6) $\Gamma' \vdash M' : A'$ such that $M' : A'$ is derivable in Γ_∞. This specific form $\Gamma' \vdash M' : A'$ we call a *type annotated* judgment.

Definition 10. *A* type annotated context *in a PTS is a context of the form*

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

where we moreover assume that all bound variables in the B_i are of the form $\dot{x}$.

Definition 11. *A* type annotated judgment *in a PTS is one of the form*

$$x_1^{B_1} : B_1, \ldots, x_n^{B_n} : B_n \vdash M : A$$

where: $x_1^{B_1} : B_1, \ldots, x_n^{B_n} : B_n$ is a type annotated context; all free variables in M and A are of the form $x_i^{B_i}$; and all bound variables are of the form $\dot{x}$.

We now first show the easy direction of our correspondence result:

Lemma 12. *Every judgment $\Gamma \vdash M : A$ in a PTS is α-equivalent to a type annotated judgment $\Gamma' \vdash M' : A'$.*

Proof. From left to right we rename the variables in the context (and of course also in M and A) to the 'standard' names

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

(Here x_i is not a meta-variable for a variable name, but really the *explicit* variable name "x_i" in $\mathscr{X}$.) We also α-rename any bound variable of the form x^A to a fresh variable of the form $\dot{x}$. $\square$

As an example, consider the PTS judgment

$$A : *,\, a : A \vdash (\lambda x{:}A.\, x)\, a : A$$

This does not fit the variable names from our $\mathscr{V}$, so this does not conform to the definitions in this paper. Instead using our variables it should be something like:

$$\dot{A} : *,\, a^* : \dot{A} \vdash (\lambda x^{\dot{B}}{:}\dot{A}.\, x^{\dot{B}})\, a^* : \dot{A}$$

This of course is not a *type annotated* judgment, the annotations make no sense at all, but still this is a perfectly fine PTS judgment as defined in Definition 1.

Now according to the theorem this is α-equivalent to a judgment that *is* type annotated. And it is, for example it is α-equivalent to

$$x_1^* : *,\, x_2^{x_1^*} : x_1^* \vdash (\lambda \dot{x}{:}x_1^*.\, \dot{x})\, x_2^{x_1^*}$$

Or, if one uses more readable names, to

$$A^* : *,\, a^{A^*} : A^* \vdash (\lambda \dot{x}{:}A^*.\, \dot{x})\, a^{A^*}$$

The other part of the easy direction of our correspondence is that a type annotated PTS judgment essentially is the same as a Γ_∞ judgment:

Theorem 13. *If the type annotated PTS judgment $\Gamma \vdash M : A$ is derivable, then $M : A$ is a derivable judgment of the corresponding Γ_∞ type theory.*

Proof. By induction on the size of the derivation of $\Gamma \vdash M : A$. We do a case split on the last rule used in the derivation:

(sort) Immediate.

(weak) Trivial, because if $\Gamma, x{:}A$ is a type annotated context, then certainly Γ is a type annotated context.

(var) By induction we have $A : s$ derivable in Γ_∞, and because the context is type annotated the variable name must be of the form x^A. Hence $x^A : A$ in Γ_∞.

(conv) We know that $\Gamma \vdash M : A$ and $\Gamma \vdash B : s$. Now A need not have bound variables of the form $\dot{x}$, but one can rename them to obtain $A' \equiv A$ such that $\Gamma \vdash M : A'$ *will* be type annotated. Then $M : A'$ and $B : s$ in Γ_∞ by induction and therefore also $M : B$ in Γ_∞.

(app) Again, we might need to change the bound variable in $\Gamma \vdash M : \Pi x{:}A.B$ to the dotted kind, to get a type annotated judgment. Apart from that this case is trivial, just like the previous one.

(Π) The conclusion will be of the form $\Gamma \vdash \Pi\dot{x}{:}A.B : s_3$. Now if we take y^A a completely fresh variable, then $\Gamma, y^A : A \vdash B[\dot{x} := y^A] : s_2$ will be a type annotated judgment, as well as being α-equivalent to $\Gamma, \dot{x} : A \vdash B : s_2$. Accordingly, let $B' := B[\dot{x} := y^A]$, so that $B \equiv B'[y^A := \dot{x}]$.

Clearly now $y^A \notin \mathrm{hfvT}(B')$ because $\Gamma, y^A : A \vdash B' : s_2$, so all type annotations will be typable in Γ, which does not contain y^A.

By induction $A : s_1$ and $B' : s_2$ in Γ_∞ and because $y^A \notin \mathrm{hfvT}(B')$ we get that $\Pi\dot{x}{:}A.B'[y^A := \dot{x}] : s_3$ in Γ_∞. But this is precisely $\Pi\dot{x}{:}A.B : s_3$.

(λ) This case essentially follows that of the previous one.

$\square$

The other direction of our correspondence—from Γ_∞ to traditional PTS style—is a bit more involved. We need to *synthesise* an appropriate context, and because we build this context by recursion over the type derivation, we need to merge these synthesised contexts using $\ltimes$. For this we need a number of lemmas involving type annotated contexts in PTSs.

Lemma 14. *If Γ and Δ are type annotated contexts, then $\Gamma \mid\mid\mid \Delta$.*

Proof. If $x^A \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)$, then $x^A : A \in \Gamma$ and $x^A : A \in \Delta$, of course, $A \equiv A$ and this is what we need to prove according to Definition 4. $\square$

Lemma 15. *If Γ is a type annotated context with $x^A \in \mathrm{dom}(\Gamma)$, $\Gamma \vdash M : B$ and $x^A \notin \mathrm{hfvT}(M,B)$, then*

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

Proof. Write $\Gamma = \Gamma_1, x^A : A, \Gamma_2$, and suppose $y^C \in \mathrm{dom}(\Gamma_2)$ with $x^A \in \mathrm{FV}(C)$. If $y^C \in \mathrm{FV}(M,B)$, then $x^A \in \mathrm{hfvT}(M,B)$, contradiction. So $y^C \notin \mathrm{FV}(M,B)$. This means that all declarations $y^C : C$ in Γ_2 for which $x^A \in \mathrm{FV}(C)$ can be removed by Strengthening (Proposition 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 $\Gamma_2' \subseteq \Gamma_2$ and $x^A \notin \mathrm{type}(\Gamma_2')$. Using Permutation (Proposition 3), we conclude that $\Gamma_1, \Gamma_2', x^A : A \vdash M : B$ and we take Γ_1, Γ_2' for Δ. $\square$

Corollary 16. *If $\Gamma \vdash M : B$ is a type annotated judgment, there is a $\Delta \subseteq \Gamma$ such that $\Delta \vdash M : A$ and* $\mathrm{dom}(\Delta) \subseteq \mathrm{hfv}(M,B)$.

Proof. Let $x^A \in \mathrm{dom}(\Gamma)$ and $x^A \notin \mathrm{hfv}(M,B)$. Then $x^A \notin \mathrm{hfvT}(M,B)$, so (according to Lemma 15), there is a $\Delta \subseteq \Gamma$ such that $\Delta, x^A : A \vdash M : B$. But also $x^A \notin \mathrm{FV}(M,B)$, so by Strengthening (Proposition 3), $\Delta \vdash M : B$. $\square$

So, in $\Gamma \vdash M : B$, we can always make the context Γ so small that its domain is within the set of hereditarily free variables of M,B. The other way around, the hereditarily free variables of M,B should be in $\mathrm{dom}(\Gamma)$:

Lemma 17. *If $\Gamma \vdash M : B$ is type annotated, then* $\mathrm{hfv}(M,B) \subseteq \mathrm{dom}(\Gamma)$.

Proof. We prove $\Gamma \vdash M : B \Rightarrow \mathrm{hfv}(M) \subseteq \mathrm{dom}(\Gamma)$, by induction on the derivation and then we are done, because if $\Gamma \vdash M : B$, then $\Gamma \vdash B : s$ for some sort s, or B is a sort.

(sort) Immediate.

(var) By induction, $\mathrm{hfv}(A) \subseteq \mathrm{dom}(\Gamma)$, so $\mathrm{hfv}(x^A) \subseteq \mathrm{dom}(\Gamma, x^A : A)$.

(conv) By induction, $\mathrm{hfv}(M) \subseteq \mathrm{dom}(\Gamma)$, so we are done.

(app) By induction, $\mathrm{hfv}(F) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{hfv}(M) \subseteq \mathrm{dom}(\Gamma)$, so $\mathrm{hfv}(F\,M) \subseteq \mathrm{dom}(\Gamma)$.

(Π) By induction, $\mathrm{hfv}(A) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{hfv}(B) \subseteq \mathrm{dom}(\Gamma, y^A : A)$, so $\mathrm{hfv}(\Pi \dot{x}{:}A.B[y^A := \dot{x}]) \subseteq \mathrm{dom}(\Gamma)$.

(λ) By induction, $\mathrm{hfv}(M) \subseteq \mathrm{dom}(\Gamma, y^A : A)$ and $\mathrm{hfv}(\Pi \dot{x}{:}A.B[y^A := \dot{x}]) \subseteq \mathrm{dom}(\Gamma)$, so $\mathrm{hfv}(A) \subset \mathrm{dom}(\Gamma)$, and hence $\mathrm{hfv}(\lambda \dot{x}{:}A.M[y^A := \dot{x}]) \subseteq \mathrm{dom}(\Gamma)$.

$\square$

The more difficult direction of our correspondence now follows:

Lemma 18. *Let $M : A$ be a derivable Γ_∞ judgment. Then all free variables in M and A have the form x^A and all bound variables have the form $\dot{x}$.*

Proof. By induction on the derivation of $M : A$. □

Theorem 19. *Let $M : A$ be a derivable Γ_∞ judgment. Then there is a type annotated judgment $\Gamma \vdash M : A$ derivable in the associated PTS, such that Γ contains exactly the variables in* $\mathrm{hfv}(M) \cup \mathrm{hfv}(A)$.

Proof. By induction on the derivation of $M : A$ we show there exists a type annotated context $\Gamma(M,A)$ such that $\Gamma(M,A) \vdash M : A$. Note that $\Gamma(M,A)$ depends on the *derivation* of the judgment $M : A$, not just on the terms M and A.

(sort) Immediate.

(var) By induction, $\Gamma(A,s) \vdash A : s$. So $\Gamma(A,s), x^A : A \vdash x^A : A$.

(conv) By induction, $\Gamma(M,A) \vdash M : A$ and $\Gamma(B,s) \vdash B : s$, and we also know that $A =_\beta B$.
So $\Gamma(M,A) \ltimes \Gamma(B,s) \vdash M : B$ by Thinning and the (conv) rule.

(app) By induction, $\Gamma(F, \Pi\dot{x}{:}A.B) \vdash F : \Pi\dot{x}{:}A.B$ and $\Gamma(M,A) \vdash M : A$.
So $\Gamma(F, \Pi\dot{x}{:}A.B) \ltimes \Gamma(M,A) \vdash F\,M : B[\dot{x} := M]$ by Thinning and the (app) rule.

(Π) By induction, $\Gamma(A,s_1) \vdash A : s_1$ and $\Gamma(B,s_2) \vdash B : s_2$.
If $y^A \notin \Gamma(B,s_2)$, then $\Gamma(A,s_1) \ltimes \Gamma(B,s_2), y^A : A_i \vdash B : s_2$, so by Thinning and the (Π) rule we have $\Gamma(A,s_1) \ltimes \Gamma(B,s_2) \vdash \Pi\dot{x}{:}A.B[y^A := \dot{x}] : s_3$.
If $y^A \in \mathrm{dom}(\Gamma(B,s_2))$, then $\Delta, y^A : A \vdash B : s_2$ for some $\Delta \subset \Gamma(B,s_2)$. So by Thinning and the (Π) rule we have $\Gamma(A,s_1) \ltimes \Delta \vdash \Pi\dot{x}{:}A.B[y^A := \dot{x}] : s_3$.

(λ) By induction, we obtain $\Gamma(M,B) \vdash M : B$ and $\Gamma(\Pi\dot{x}{:}A.B[y^A := \dot{x}], s) \vdash \Pi\dot{x}{:}A.B[y^A := \dot{x}] : s$.
If $y^A \notin \mathrm{dom}(\Gamma(M,B))$, then $\Gamma(A,s_1) \ltimes \Gamma(M,B), y^A : A_i \vdash M : B$. So by Thinning and the (λ) rule, we have $\Gamma(\Pi\dot{x}{:}A.B[y^A := \dot{x}], s) \ltimes \Gamma(M,B) \vdash \lambda\dot{x}{:}A.M[y^A := \dot{x}] : \Pi\dot{x}{:}A.B[y^A := \dot{x}]$.
If $y^A \in \mathrm{dom}(\Gamma(M,B))$, then $\Delta, y^A : A \vdash M : B$ for some $\Delta \subset \Gamma(M,B)$, by Lemma 15. So by Thinning and the (λ) rule, we have $\Gamma(\Pi\dot{x}{:}A.B[y^A := \dot{x}], s) \ltimes \Delta \vdash \lambda\dot{x}{:}A.M[y^A := \dot{x}] : \Pi\dot{x}{:}A.B[y^A := \dot{x}]$.

By Lemma 17, $\mathrm{dom}(\Gamma(M,A)) \supseteq \mathrm{hfv}(M,A)$. Corollary 16 lets us strengthen the context to $\Delta \subseteq \Gamma(M,A)$ such that $\Delta \vdash M : A$ and $\mathrm{dom}(\Delta) = \mathrm{hfv}(M,A)$. □

Corollary 20. *Let*

$$M : A$$

be a derivable Γ_∞ judgment. Take all variables of the form x_i^A occurring in $\mathrm{hfv}(M) \cup \mathrm{hfv}(A)$, *and put them in* any *order $x_{i_1}^{A_1}, \ldots, x_{i_n}^{A_n}$ such that if $x_{i_k}^{A_k}$ occurs in A_l then $k < l$. Then the following judgment is derivable in the PTS:*

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

Proof. From the previous Theorem using Permutation. □

5 Conclusion and Further work

There are three obvious continuations of this work:

1. The first is to investigate to what extent 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.
3. The third is to formally develop the theory presented in this paper in a proof assistant.

With respect to the first point: we expect most type theories to have a Γ_∞-variant, although the observations about universe inconsistency [19] arising from merging contexts may complicate the picture for applied type systems such as that of Coq. More important is to investigate how our approach needs to be adapted to support type theories with definitions. As previously noted, in any real implementation, the definitions for defined constants form a more significant part of the contexts Γ we are eliminating than the free variables.

We are currently investigating the second point, developing an OCaml implementation for the PTS λP extended with definitions (a system corresponding to the logical framework LF) along the lines of this paper. The main question is how expensive, computationally, the two following operations are:

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

The first is in some sense 'local', because it does not look inside constant definitions in the environment. To make the second reasonably efficient will be harder. It is possible we need to consider *three* kinds of variables, distinguishing (1) $\dot{x}$ bound variables, (2) y^A variables to be substituted with bound variables later (essentially, the *eigenvariables* of the (Π) and (λ) rules), and (3) x^A variables that do remain free, so they may be considered as 'axiomatic constants' of the system.

In such a system, it is essential that the implementation language can use pointer equality to efficiently determine equality of terms (and in particular, equality of annotations in variable occurrences). This motivated our choice of OCaml, which is such a language. Although in OCaml the comparison function "`=`" does not have this feature (because floating points NaNs are not taken to be equal to themselves, the system never looks at pointer equality when evaluating "`x = y`"), the comparison function "`fun x y -> Pervasives.compare x y = 0`" does.

We are currently working on the third point as well. In ongoing work [10] the second author has formalised a large part of the theory presented in this paper up to the first direction of the correspondence theorem in the proof assistant Coq. However, the presentation in this formal development is slightly different from that of this paper.

- Firstly, we distinguish bound from free variables at the level of PTS pseudo-terms, following existing practice, established since the third author's work with Pollack in the 1990s [14, 15]. Our informal presentation above uses named variables in each case, the so-called *locally named* approach, whereas the formalisation uses the *locally nameless* representation: de Bruijn indices for bound variables and names for free variables. Because we make this difference at the level of PTS pseudo-terms already, we have a canonical representative for each term and therefore need not worry about α-equivalence. For further details of both approaches, see [15, 1, for example].
- Secondly, variable binding in the (Π) and (λ) rules of Γ_∞ is handled by substituting free variables for bound variables, rather than bound for free as in Section 3. This choice has been used successfully in other formalisations (*ibid.*), and emphasises the conceptual priority of free variables over

bound. Recent work by Pollack and Sato compares the two approaches, in a detailed account of canonical representation for languages with binding [20].

Based on the methodology described in [1], we have combined the locally nameless presentation with co-finite quantification to obtain strong induction principles. To be sure that our co-finite presentation is adequate we have proved it to be equivalent to an exists-fresh presentation.

For example the (Π) rule, in respectively exists-fresh and co-finite presentation, is as follows.

$$(\Pi\text{ exists-fresh})\quad \frac{A : s_1 \quad B[0 := x^A] : s_2}{\Pi A.B : s_3} \qquad \text{if } (s_1, s_2, s_3) \in \mathscr{R} \text{ and } x^A \notin \mathrm{hfv}(B)$$

$$(\Pi\text{ co-finite})\quad \frac{A : s_1 \quad \forall x \notin L\ .\ B[0 := x^A] : s_2}{\Pi A.B : s_3} \quad \text{if } (s_1, s_2, s_3) \in \mathscr{R} \text{ and } L \subset_{\text{finite}} \mathscr{V}$$

Observe that we require $x^A \notin \mathrm{hfv}(B)$ instead of $x^A \notin \mathrm{hfvT}(B)$ (this was also observed by John Boyland). The reason for this is that we have to bind every occurrence of the free variable x^A in $B[0 := x^A]$, hence x^A should not be in $\mathrm{FV}(B)$ either. While the condition is (potentially) more expensive to check, it removes the need for Definition 8, in favour of the (conceptually) simpler Definition 7.

The other direction of the correspondence theorem uses the second property, Strengthening, of Proposition 3 in an essential way. This presents two difficulties: a practical one, since the existing formalisations of this lemma are highly non-trivial [15]; and a theoretical one, namely that we may *not* be able to establish a correspondence between traditional and Γ_∞ presentations of a given type theory without first establishing strengthening.

More interestingly, from the point of view of the pragmatics of formalisation, for this direction it is essential to abstract over the kinds of free variables used in the definition of PTS judgments. At first this does not seem troublesome, however, many definitions and theorems do not just depend on the kind of free variables but also on finite sets of free variables. Hence we are also required to abstract over various operations on such finite sets. The recently developed finite set library [13], based on the new type classes feature in Coq, might be very useful in implementing this abstraction.

Acknowledgments Thanks to Jean-Christophe Filliâtre for details about the architecture of the Coq kernel. We are grateful to the anonymous referees, and to the audience at LFMTP, especially John Boyland, Brigitte Pientka and Andrew Pitts, who made several helpful remarks, especially concerning related work. This research is partially funded by the NWO BRICKS/FOCUS project "ARPA: Advancing the Real use of Proof Assistants" and the NWO cluster DIAMANT.

References

[1] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack & S Weirich (2008): *Engineering Formal Metatheory*. In: *POPL'08*, ACM, pp. 3–15.

[2] H. Barendregt (1992): *Lambda Calculi with types*. In: S. Abramsky, Dov M. Gabbay & T.S.E. Maibaum, editors: *Handbook of Logic in Computer Science, Volume 2*, OUP, pp. 117–309.

[3] H. Barendregt & H. Geuvers (2001): *Proof Assistants using Dependent Type Systems*. In: A. Robinson & A. Voronkov, editors: *Handbook of Automated Reasoning*, Elsevier, pp. 1149–1238.

[4] M. Boespflug (2010): *Context-free Pure Type Systems*. Submitted to POPL.

[5] A. Church (1940): *A Formulation of the Simple Theory of Types*. *J. Symbolic Logic* 5, pp. 56–68.

[6] N.G. de Bruijn (1979): *Wees contextbewust in WOT*. *Euclides* 55, pp. 7–12.

[7] H. Geuvers (1993): *Logics and Type Systems*. Ph.D. thesis, Nijmegen University.

[8] J-Y. Girard (1972): *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII.

[9] M. Gordon, R. Milner & C. Wadsworth (1979): *Edinburgh LCF: A mechanised logic of computation*, *LNCS* 78. Springer.

[10] R. Krebbers (2010). *A formalization of* Γ_∞ *in Coq*. `http://robbertkrebbers.nl/research/gammainf/`.

[11] J. Lambek & P. Scott (1988): *Introduction to Higher-Order Categorical Logic*. Number 7 in Cambridge Studies in Advanced Mathematics. CUP.

[12] K. Lambert (1963): *Existential import revisited*. *Notre Dame J. Formal Logic* 4(4), pp. 288–292.

[13] S. Lescuyer (2010). *Containers*. `http://www.lri.fr/~lescuyer/Containers.en.html`.

[14] J. McKinna & R. Pollack (1993): *Pure type systems formalized*. In: J-F.Groote & M. Bezem, editors: *TLCA'93*, *LNCS* 664, Springer, pp. 289–305.

[15] J. McKinna & R. Pollack (1999): *Some lambda calculus and type theory formalized*. *J. Automated Reasoning* 23, pp. 373–409.

[16] E. Mendelson (1964): *Introduction to Mathematical Logic*. van Nostrand.

[17] R. Pollack (1994): *Closure under alpha-conversion*. In: *TYPES '93*, *LNCS* 806, Springer, pp. 313–332.

[18] C. Sacerdoti Coen (2004): *Knowledge Management of Formal Mathematical Theories and Interactive Theorem Proving*. Ph.D. thesis, University of Bologna. Technical report UBLCS-2004-05.

[19] C. Sacerdoti Coen (2004): *Mathematical Libraries as Proof Assistant Environments*. In: A. Asperti, G. Bancerek & A. Trybulec, editors: *MKM2004*, *LNCS* 3119, Springer, pp. 332–346.

[20] M. Sato & R. Pollack (2010): *Internal and External Syntax of the* λ*-calculus*. *J. Symbolic Computation* 45, pp. 598–616.

[21] P. Severi & E. Poll (1994): *Pure Type Systems with definitions*. In: *Logical Foundations of Computer Science '94*, *LNCS* 813, Springer, pp. 316–328.

[22] D. van Dalen (1980): *Logic and Structure*. Springer.

[23] J.A. Vaughan (2006). *A Review of Three Techniques for Formally Representing Variable Binding*. University of Pennsylvania CIS Technical Report Number MS-CIS-06-19.

[24] F. Wiedijk (2009): *Stateless HOL*. In: J.W. Klop, V. van Oostrom & F. van Raamsdonk, editors: *Liber Amicorum for Roel de Vrijer*, Vrije Universiteit Amsterdam, pp. 227–240.