A Formalization of the Theory of Pure Type Systems in Coq

Robin Adams

University of Manchester
`radams@cs.man.ac.uk`

Abstract. We describe a recent formalization of several results from the metatheory of Pure Type Systems (PTSs) in Coq, including Subject Reduction, Uniqueness of Types in a functional PTS, and the difficult proof of Strengthening. The terms of the PTS are represented by an inductive family of types: "`term` n" is the type of all terms with at most n free variables. This representation of terms has often been used to study syntax and substitution, but not the metatheory of a formal system. We show how it requires many metatheorems to be stated in a somewhat unfamiliar "big-step" form, but then allows for very elegant and direct proofs.

1 Introduction

About six months ago, late in 2004, I decided to formalise a technically complex proof of a result of mine in the theory of Pure Type Systems (PTSs) [1], in order to guarantee its correctness. I looked around for formalizations of the metatheory of PTSs or similar type systems that I could use or adapt. I found two major developments [2, 3], and was dissatisfied with both of them. To use either would have involved spending much time on technical details that were irrelevant to the main proof ideas. While this is inevitable to some extent whenever one formalises a piece mathematics, it seemed worth spending some time searching for another approach with which the technicalities would be fewer.

There came to my mind an idea I had first been introduced to in a talk by Peter Aczel [4]: that of representing the terms of the lambda calculus, or any syntax with binding, as a *nested data type*. I decided to experiment with a formalization based on such a representation of the grammar. To my delight, I found myself making very rapid progress: the technicalities involved were indeed fewer, or at least were better handled by Coq's automation.

The formalization often dealt more easily with what I came to call "big-step" operations and theorems, as opposed to the "small-step" form which was usually more familiar. For example, I quickly found that it is much easier to define the operation of substituting for all variables simultaneously, rather than for a single variable. The Substitution Lemma was also more easily proven when stated in these terms. So some thought was often necessary as to the best form for a definition or theorem to take; however, once the correct form for a theorem had been found, the proof usually proceeded very quickly, simply and directly.

Moreover, the formalization had to my mind a more type-theoretic flavour than either of the two cited above: the objects with which one was dealing were objects one might reasonably expect to find when dealing with the metatheory of a type theory, rather than low-level aspects of the mechanisms of the formalization.

Unfortunately, after formalizing many results in the basic theory of PTSs, I was not able to spare the time to complete the formal proof of the result in [1], although I do plan to return to this task when time permits. However, I hope that the work I have completed so far will be of interest to others — either those who wish a formalization of PTSs they can use, or those who wish to build a similar formalization of a different system. There may also be some independent interest in seeing what form the results in the metatheory of a formal system take when one represents the grammar of the system as a nested data type.

We shall not be too strict in this paper with accuracy when giving the details of the formalization: we shall freely mix Coq keywords with informal mathematical notation, and shall reverse the order of the arguments of a function if we feel it aids readability. (For example, the type we here denote by `string` $m\ n$ would be the type `string` $n\ m$ in the formalization.) This paper is a guide to the ideas used in the formalization, not a documentation of the formalization itself.

For those who wish to examine the formalization, the source code and documentation is available at `http://www.cs.man.ac.uk/~radams/coqPTS`.

2 Pure Type Systems

The theory of Pure Type Systems (PTSs) is a theory dealing with a large number of type theories — it is the closest thing yet developed to a general theory of type theories. It is not too much of an exaggeration to say that any theory whose only type constructor is Π can be defined as a PTS: this includes the simply-typed lambda calculus, the Calculus of Constructions, System F, and so forth.

The theory of PTSs is an excellent example of a beautiful mathematical theory. It begins with a simple set of definitions, from which follow a rich body of theorems, some basic, some deep. The way that a PTS is specified could not be simpler: a set, with a binary and ternary relation. The set of terms is then given by a grammar with just five constructors, and the derivable judgements given by just seven rules of deduction.

The best reference for the theory of PTSs is still Barendregt [5]. The formalization follows the development in this paper very closely. We prove all the results given there for arbitrary PTSs, from Lemma 5.2.8 to Corollary 5.2.22. (Lemma 5.2.7, the Free Variables Lemma, follows automatically from the way we define syntax.) We also prove the Strengthening Lemma, following the proof given in van Bentham Jutting [6].

3 Terms as a Nested Data Type

The idea of representing terms as a nested data type first appeared in Bellegarde and Hook [7]. It has been used several times for the study of syntactic properties and operations, such as substitution and β-reduction; see, for example, Altenkirch and Reus [8] or Bird and Paterson [9], the latter of which even deals with the simply-typed lambda calculus. However, this representation of syntax seems never before to have been used in the metatheory of a typing system where the typing judgements are given by a set of rules of deduction.

We shall briefly summarize the ideas behind this representation here; more details can be found in the three papers cited above. We wish to represent, in some type theory, the terms of some formal system whose syntax involves one or more binding constructor. Rather than defining one type whose objects are to represent these terms, we define a family of types

$$\mathcal{T}_V$$

for every type V in some universe. The objects of type $\mathcal{T}_V$ represent those terms that can be formed using the objects of type V as free variables.

For example, let us assume we have defined, for every type X, a type $X_\perp$ consisting of a copy $\uparrow x$ of each $x : X$, together with one extra object, $\perp$. We can then represent the terms of the untyped lambda calculus by the inductive family $\mathcal{T}_V$ whose constructors are as follows.

$$\frac{x : V}{\mathtt{var}\ x : \mathcal{T}_V} \qquad \frac{M : \mathcal{T}_{V_\perp}}{\lambda M : \mathcal{T}_V} \qquad \frac{M : \mathcal{T}_V \quad N : \mathcal{T}_V}{MN : \mathcal{T}_V}$$

The constructor λ takes a term M whose free variables are taken from $V_\perp$, binds the variable $\perp$, and returns a term λM whose free variables are taken from V.

The definition of substitution takes the following form. We aim to define, for every term $M : \mathcal{T}_U$ and every *substitution function* $\sigma : U \to \mathcal{T}_V$, the term $M[\sigma] : \mathcal{T}_V$, the result of substituting for *each* variable $u : U$ the term $\rho(u) : \mathcal{T}_V$. This is our first encounter with a theme that shall appear many times in this formalization: it is easier to deal with every variable in a given type V at once ("big-step" operations and theorems) than with a single variable.("small-step" operations and theorems). The small-step version can always be recovered later as a special case of the big-step version.

The definition of substitution that we would naturally write down is as follows.

$$\begin{aligned} x[\sigma] &\equiv \sigma(x) \\ (MN)[\sigma] &\equiv M[\sigma]N[\sigma] \\ (\lambda M)[\sigma] &\equiv \lambda M \begin{bmatrix} \perp \mapsto \perp \\ \uparrow x \mapsto \sigma(x)[y \mapsto \uparrow y] \end{bmatrix} \end{aligned}$$

(Here and henceforth, we are omitting the constructor `var`.)

It is possible to prove that this recursion terminates. However, this definition cannot be made directly in Coq, as it is not a definition by structural recursion. But we can define the special case where ρ always returns a variable. We thus define the operation of *replacement*; given a term $M : \mathcal{T}_U$ and a *renaming function* $\rho : U \to V$, we define the term $M\{\rho\} : \mathcal{T}_V$, the result of replacing each variable $u : U$ with $\rho(u) : V$, thus:

$$\begin{aligned} x\{\rho\} &\equiv \rho(x) \\ (MN)\{\rho\} &\equiv M\{\rho\}N\{\rho\} \\ (\lambda M)\{\rho\} &\equiv \lambda M \left\{ \begin{matrix} \perp \mapsto \perp \\ \uparrow x \mapsto \uparrow \rho(x) \end{matrix} \right\} \end{aligned}$$

Substitution can then be defined as follows: for $\sigma : U \to \mathcal{T}_V$,

$$\begin{aligned} x[\sigma] &\equiv \sigma(x) \\ (MN)[\sigma] &\equiv M[\sigma]N[\sigma] \\ (\lambda M)[\sigma] &\equiv \lambda M \left[\begin{matrix} \perp \mapsto \perp \\ \uparrow x \mapsto \rho(x)\{y \mapsto \uparrow y\} \end{matrix} \right] \end{aligned}$$

Altenkirch and Reus [8] show how the type operation $\mathcal{T}$, the variable constructor `var`, and the substitution operation [ ] form a *Kleisli triple*, a concept closely related to that of a monad. Bird and Paterson [9] give the monadic structure explicitly: the substitution operation can be split into a mapping $\mathcal{T}_U \to \mathcal{T}_{\mathcal{T}_V}$, followed by a "folding" operation $\mathcal{T}_{\mathcal{T}_V} \to \mathcal{T}_V$; the type operator $\mathcal{T}$, together with the constructor `var`, the replacement operation { } and this folding operation, form a monad.

For this formalization, we modify this constuction slightly. Rather than allowing any type V : `Set` to be used as the type of variables, we only use the members of a family $\mathcal{F}_n$ of finite types, $\mathcal{F}_n$ having n distinct canonical members. We then define the type $\mathcal{T}_n$ of terms that use the objects of $\mathcal{F}_n$ as free variables. We shall refer to this representation of syntax as "`nat`-indexed terms". As the objects of $\mathcal{F}_n$ come in a canonical order, the result syntax is very close to de Bruijn notation.

4 Description of the Formalization

4.1 Grammar

We define a family of finite types $\mathcal{F}_n$ $(n : \mathbb{N})$, $\mathcal{F}_n$ having exactly n distinct canonical objects. This could be defined directly as an inductive family:

$$\frac{n : \mathbb{N}}{\perp : \mathcal{F}_{n+1}} \qquad \frac{x : \mathcal{F}_n}{\uparrow x : \mathcal{F}_{n+1}}$$

However, it was found that case analysis on an object of type $\mathcal{F}_n$ is easier in Coq if $\mathcal{F}$ is instead defined as a recursive function thus:

$$\begin{aligned} \mathcal{F}_0 &= \emptyset \\ \mathcal{F}_{n+1} &= (\mathcal{F}_n)_\perp \end{aligned}$$

Here, $\emptyset$ (`empty`) is the empty type, and $X_\perp$ (`option` X) is a type consisting of a copy $\uparrow x$ of each object $x : X$ together with one extra object $\perp$; both of these are provided by the Coq library. (In general, when working with Coq, it seems to be advisible to use several simple inductive definitions rather than one complex one. We shall be faced with a similar choice when we come to define the types of contexts.)

We then define the family of types $\mathcal{T}_n$ of terms using the objects of type $\mathcal{F}_n$ as free variables, for $n : \mathbb{N}$:

$$\frac{x : \mathcal{F}_n}{x : \mathcal{T}_n} \qquad \frac{s : \mathcal{S}}{s : \mathcal{T}_n} \qquad \frac{M : \mathcal{T}_n \quad N : \mathcal{T}_n}{MN : \mathcal{T}_n} \qquad \frac{A : \mathcal{T}_n \quad B : \mathcal{T}_{n+1}}{\Pi AB : \mathcal{T}_n} \qquad \frac{A : \mathcal{T}_n \quad M : \mathcal{T}_{n+1}}{\lambda AM : \mathcal{T}_n}$$

We proceed to define the replacement operation $\{\ \}$ and the substitution operation $[\,]$, as discussed in the previous section. We can then prove the following versions of the Substitution Lemma.

Lemma 1. *Let $M : \mathcal{T}_m$; $\rho : \mathcal{F}_m \to \mathcal{F}_n$; $\rho' : \mathcal{F}_n \to \mathcal{F}_p$; $\sigma : \mathcal{F}_m \to \mathcal{T}_n$; and $\sigma' : \mathcal{F}_n \to \mathcal{T}_p$. Then*

$$\begin{aligned} M\{\rho\}\{\rho'\} &\equiv M\{\rho' \circ \rho\} \\ M\{\rho\}[\sigma'] &\equiv M[\sigma' \circ \rho] \\ M[\sigma]\{\rho'\} &\equiv M[x \mapsto \sigma(x)\{\rho'\}] \\ M[\sigma][\sigma'] &\equiv M[x \mapsto \sigma(x)[\sigma']] \end{aligned}$$

4.2 Reduction and Conversion

We define the relation of *parallel one-step reduction*, $\rhd$, which has type $\Pi n : \mathbb{N}.\mathcal{T}_n \to \mathcal{T}_n \to$ `Prop`, thus:

$$\frac{x : \mathcal{F}_n}{x \rhd x} \qquad \frac{s : \mathcal{S}}{s \rhd s} \qquad \frac{M \rhd M' \quad N \rhd N'}{MN \rhd M'N'}$$

$$\frac{A \rhd A' \quad B \rhd B'}{\Pi AB \rhd \Pi A'B'} \qquad \frac{A \rhd A' \quad M \rhd M'}{\lambda AM \rhd \lambda A'M'} \qquad \frac{M \rhd M' \quad N \rhd N'}{(\lambda AM)N \rhd M'[N']}$$

The relation of reduction, $\twoheadrightarrow$, is defined to be the transitive closure of $\rhd$, and the relation of convertibility, $\simeq$, is defined to be the symmetric, transitive closure of $\rhd$.

We are then able to prove that $\rhd$ satisfies the diamond property, and deduce the two forms of the Church-Rosser Theorem:

Lemma 2. *For $M, N, P : \mathcal{T}_n$, if $M \rhd N$ and $M \rhd P$, then there exists $Q : \mathcal{T}_n$ such that $N \rhd Q$ and $P \rhd Q$.*

Theorem 1. *For $M, N, P : \mathcal{T}_n$, if $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, then there exists $Q : \mathcal{T}_n$ such that $N \twoheadrightarrow Q$ and $P \twoheadrightarrow Q$.*

Theorem 2. *For $M, N : \mathcal{T}_n$, if $M \simeq N$, then there exists $P : \mathcal{T}_n$ such that $M \twoheadrightarrow P$ and $N \twoheadrightarrow P$.*

4.3 Pure Type Systems

Contexts. We now define the type $\mathcal{C}_n$ of contexts with domain $\mathcal{F}_n$. Just as for $\mathcal{F}_n$, in order to make case analysis easier, we do not define this as an inductive family, but rather by recursion on n as follows:

$$\mathcal{C}_0 = \mathbf{1}$$
$$\mathcal{C}_{n+1} = \mathcal{C}_n \times \mathcal{F}_n$$

Here, $\mathbf{1}$ (`unit`) is a type with a unique canonical element $*$ (`tt`), and $A \times B$ (`prod` A B) is the Cartesian product of A and B, both provided by the Coq library.

We can define the function `typeof`, with type $\Pi n : \mathbb{N}.\mathcal{F}_n \to \mathcal{C}_n \to \mathcal{T}_n$, which looks up the type of the variable $x : \mathcal{F}_n$ in the context $\Gamma : \mathcal{C}_n$. We write $\Gamma(x)$ for `typeof` $\_ \; x \; \Gamma$. Thus, the situation that we would describe on paper by "$x : A \in \Gamma$" is expressed in our formalization by $\Gamma(x) \equiv A$.

The `typeof` function is defined by recursion on n as follows (the case $n = 0$ being vacuous):

$$\langle \Gamma, A\rangle(\bot) \equiv A\{\uparrow\}$$
$$\langle \Gamma, A\rangle(\uparrow x) \equiv \Gamma(x)\{\uparrow\}$$

Rules of Deduction. We declare the axioms $\mathcal{A}$ and rules $\mathcal{R}$ of the arbitrary PTS with which we are working as parameters:

$$\texttt{Parameter}\ \mathcal{A} : \mathcal{S} \to \mathcal{S} \to \texttt{Prop}$$
$$\texttt{Parameter}\ \mathcal{R} : \mathcal{S} \to \mathcal{S} \to \mathcal{S} \to \texttt{Prop}$$

We are now finally able to defined the typing relation $\vdash$ of the PTS. This relation has type

$$\Pi n : \mathbb{N}.\mathcal{C}_n \to \mathcal{T}_n \to \mathcal{T}_n \to \texttt{Prop}$$

When given a context $\Gamma : \mathcal{C}_n$ and terms $M, A : \mathcal{T}_n$, it returns the proposition "The judgement $\Gamma \vdash M : A$ is derivable". It is defined as an inductive relation, and its constructors are simply the rules of deduction of a PTS (see Fig. 1).

Subcontext Relation. The definition of the subcontext relation is the first place where our formalization differs significantly from the informal

When we use named variables, the subcontext relation is defined as follows: the context Γ is a subcontext of Δ iff, for every variable x and type A, if $x : A \in \Gamma$ then $x : A \in \Delta$. We could think of a *function* giving, for each entry $x : A$ in Γ, the position at which $x : A$ occurs in Δ. (See Fig. 2 (a).)

If Γ is of length m and Δ of length n, then we can write the definition in the following form:

> Γ is a subcontext of Δ iff there exists a function $\rho : \{1, \ldots, m\} \to \{1, \ldots, n\}$ such that, if the ith entry of Γ is $x : A$, then the $\rho(i)$th entry of Δ is $x : A$.

$$\texttt{axioms}: \quad \frac{}{* \vdash s : t} \quad (\mathcal{A}\ s\ t)$$

$$\texttt{start}: \quad \frac{\Gamma \vdash A : s}{\Gamma, A \vdash \bot : A\{\uparrow\}}$$

$$\texttt{weakening}: \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, C \vdash A\{\uparrow\} : B\{\uparrow\}}$$

$$\texttt{product}: \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, A \vdash B : s_2}{\Gamma \vdash \Pi AB : s_3} \quad (\mathcal{R}\ s_1\ s_2\ s_3)$$

$$\texttt{application}: \quad \frac{\Gamma \vdash M : \Pi AB \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B \begin{bmatrix} \bot \mapsto N \\ \uparrow x \mapsto x \end{bmatrix}}$$

$$\texttt{abstraction}: \quad \frac{\Gamma, A \vdash M : B \quad \Gamma \vdash A : s_1 \quad \Gamma, A \vdash B : s_2}{\Gamma \vdash \lambda AM : \Pi AB} \quad (\mathcal{R}\ s_1\ s_2\ s_3)$$

$$\texttt{conversion}: \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \quad (A \simeq B)\ .$$

Fig. 1. Rules of deduction of a Pure Type System

In our formalization, we cannot use the same definition, as the variables of $\mathcal{F}_n$ come in a fixed order: $\bot$, $\uparrow \bot$, $\uparrow\uparrow \bot$, ... But we can still talk of functions mapping positions in one context to positions in another: these are the *renaming* functions $\rho : \mathcal{F}_m \to \mathcal{F}_n$. (See Fig. 2 (b).)

We can therefore define the relation: Γ is a subcontext of Δ *under* the renaming ρ. This means that, if we identify each variable x in Γ with the variable $\rho(x)$ in Δ, then the entry with subject x in Γ is the same as the entry with subject $\rho(x)$ in Δ. More precisely:

> Γ is a subcontext of Δ under ρ, $\Gamma \subseteq_\rho \Delta$,‘ iff, for each variable x in $\mathcal{F}_n$, if $x : A$ occurs in Γ, then $\rho(x) : A\{\rho\}$ occurs in Δ.

In terms of the `typeof` function, this becomes:

Definition 1. *Let $\Gamma : \mathcal{C}_m$, $\Delta : \mathcal{C}_n$, and $\rho : \mathcal{F}_m \to \mathcal{F}_n$. Γ is a subcontext of Δ under ρ, $\Gamma \subseteq_\rho \Delta$, iff*

$$(\forall x : \mathcal{F}_m)\Delta(\rho(x)) \equiv \Gamma(x)\{\rho\}$$

Note that the relation we have defined allows contraction: ρ may map two different variables x and y to the same variable in $\mathcal{F}_n$ (in which case x and y must have the same type in Γ). If we wish to exclude contraction, we shall use the relation "$\Gamma \subseteq_\rho \Delta \wedge \rho$ is injective".

With this definition, the Weakening Lemma becomes:

Named Variables — $\Gamma \subseteq \Delta$:

$$\forall x, A (x : A \in \Gamma \Rightarrow x : A \in \Delta)$$

$$\begin{tikzcd}
\Gamma \equiv & & x : A, \arrow[d] & y : B, \arrow[drrr] & \ldots, & z : C \arrow[dl] & \\
\Delta \equiv & \ldots, & x : A, & \ldots, & z : C, & \ldots, & y : B
\end{tikzcd}$$

`Nat`-*indexed Terms* — $\Gamma \subseteq_\rho \Delta$:

$$\forall x\ \Gamma(x)\{\rho\} \equiv \Delta(\rho(x))$$

$$\begin{tikzcd}
\Gamma \equiv & & \uparrow^n \bot : A, \arrow[d, "\rho"'] & \uparrow^{n-1} \bot : B, \arrow[drrr] & \ldots, & \bot : C \arrow[dl] & \\
\Delta \equiv & \ldots, & \rho(\uparrow^n \bot) : A\{\rho\}, & \ldots, & \rho(\uparrow^{n-1} \bot) : C\{\rho\}, & \ldots, & \rho(\bot) : B\{\rho\}
\end{tikzcd}$$

Fig. 2. The definition of subcontext

Lemma 3. *If $\Gamma \subseteq_\rho \Delta$, $\Gamma \vdash M : A$, and Δ is valid, then $\Delta \vdash M\{\rho\} : A\{\rho\}$.*

Just as we are using big-step notions of replacement and substitution, so our Weakening Lemma takes a big-step form. The corresponding small-step form is

> If $\Gamma, \Delta \vdash M : A$ and $\Gamma \vdash B : s$, then $\Gamma, x : B, \Delta \vdash M : A$ (where x is not free in Δ, M or A).

The particular instances of the small-step Weakening Lemma that we need to use in later proofs are easily derived as special cases of the big-step version.

The big-step version of the Weakening Lemma is not particularly novel: it appears in many informal and formal developments. The form that the next result takes is probably more surprising:

Substitution Lemma. What form should the Substitution Lemma take, given that we are using substitution mappings $\sigma : \mathcal{F}_m \to \mathcal{T}_n$? A moment's thought will show that, with named variables, the result would read as follows:

> If $x_1 : A_1, \ldots, x_m : A_m \vdash M : B$ and
>
> $$\Gamma \vdash \sigma(x_1) : A_1[\sigma]$$
> $$\vdots$$
> $$\Gamma \vdash \sigma(x_m) : A_m[\sigma]$$
>
> then $\Gamma \vdash M[\sigma] : B[\sigma]$.

(For the case $m = 0$, we would need to add the hypothesis "Γ is valid".)

It is now quite clear how the result should read with `nat`-indexed terms. Let us define the relation $\Gamma \models \sigma :: \Delta$ (σ *satisfies* the context Δ under Γ) to mean

$$\forall x : \mathcal{F}_m . \Gamma \vdash \sigma(x) : \Delta(x)[\sigma] \ .$$

Then our big-step Substitution Lemma reads:

Lemma 4 (Substitution). *If $\Delta \vdash M : B$, $\Gamma \models \sigma :: \Delta$, and Γ is valid, then $\Gamma \vdash M[\sigma] : B[\sigma]$.*

This is proven by induction on the derivation of $\Delta \vdash M : B$.

Subject Reduction. Subject Reduction similarly takes a big-step form. Subject Reduction is traditionally stated in the form:

If $\Gamma \vdash M : A$ and $M \twoheadrightarrow N$, then $\Gamma \vdash N : A$.

In our formalization, the most convenient form in which to prove Subject Reduction is as follows. We extend the notion of parallel one-step reduction to contexts, by making the following definition:

Definition 2. *Define the relation of* parallel one-step reduction, *$\triangleright$, on $\mathcal{C}_n$ as follows: $\Gamma \triangleright \Delta$ iff $\forall x : \mathcal{F}_n . \Gamma(x) \triangleright \Delta(x)$.*

We can now prove:

Theorem 3 (Subject Reduction). *If $\Gamma \vdash M : A$, $\Gamma \triangleright \Delta$ and $M \triangleright N$, then $\Delta \vdash N : A$.*

This is proven directly by induction on $\Gamma \vdash M : A$. The usual form of Subject Reduction follows quite simply.

Other Results. The formalization also contains proofs of Context Conversion, the Generation lemmas, Type Validity, Predicate Reduction and the Uniqueness of Types result for functional PTSs. There is little to say about any of these results: Context Conversion takes the expected big-step form, while the others take the same form, and the proofs follow the same lines, as the paper development.

4.4 The Proof of the Strengthening Theorem

The proof of the Strengthening Theorem in van Bentham Jutting's paper [6] is a technically complex proof, and a very good, tough test of any formalization of the theory of PTSs.

Perhaps surprisingly, the feature that makes the proof most difficult to formalise when using `nat`-indexed terms is the use of terms of the form

$$\Lambda\Delta.M \text{ and } \Pi\Delta.M$$

where Δ is a *string* of abstractions

$$\Delta \equiv \langle x_1 : A_1, \ldots, x_n : A_n \rangle$$

We shall need to build the type of strings in such a way that these operations Λ and Π can be defined. A closer look at the proofs in [6] reveals that we shall also need an operation for concatenating a context with a string; that is, given a context Γ and a string Δ, producing a context (Γ, Δ). We must also be able to apply a substitution to a string.

Several different approaches to these definitions were tried before the ones described below were found. As a general principle, it was found to be important to *avoid arithmetic within types* at all costs. Once addition appeared within the type of some operator, one inevitably found oneself, sooner or later, in some impossible situation, such as requiring a term to have type $\mathcal{T}_{m+\mathtt{S}n}$ at one point in a formula and type $\mathcal{T}_{\mathtt{S}m+n}$ at another point in the same formula.

For example, there are zeveral possible ways to define the family of types of strings. We could define

- $\mathtt{string1}\ m\ p$ — the type of all strings of length p whose initial term has type $\mathcal{T}_m$
- $\mathtt{string2}\ n\ p$ — the type of all strings of length p whose final term has type $\mathcal{T}_n$
- $\mathtt{string3}\ m\ n$ — the type of all strings whose initial term has type $\mathcal{T}_m$ and whose final term has type $\mathcal{T}_{n-1}$. (It follows necessarily that $m \leq n$ and the string has length $n - m$.)

Only with the third choice can we avoid arithmetic in the type of the Π and Λ operators. For example, if we chose to use `string1`, the Π operator would need to have the type

$$\Pi m, p : \mathbb{N}. \mathtt{string1}\ m\ p \to \mathcal{T}_{m+p} \to \mathcal{T}_m \quad .$$

With substitution, it is far more difficult to find a set of definitions that avoids the need for addition within some type. A subtle solution was eventually found, and is described below.

Strings. We define the inductive family of types `string`. If $n \leq m$, the type $\mathtt{string}\ m\ n$ is the type of all strings

$$\Delta \equiv \langle A_m, A_{m+1}, A_{m+2}, \ldots, A_{n-1} \rangle$$

such that $A_m : \mathcal{T}_m$, $A_{m+1} : \mathcal{T}_{m+1}$, ..., $A_{n-1} : \mathcal{T}_{n-1}$. If $n = m$, the type.has only one member (the empty string); and if $n > m$, $\mathtt{string}\ m\ n$ is empty.

`Inductive string`

$$\frac{m : \mathbb{N} \quad n : \mathbb{N}}{\mathtt{string}\ m\ n : \mathtt{Set}} \qquad \frac{n : \mathbb{N}}{\langle\rangle : \mathtt{string}\ n\ n} \qquad \frac{A : \mathcal{T}_m \quad \Delta : \mathtt{string}\ m+1\ n}{A :: \Delta : \mathtt{string}\ m\ n}$$

We can define destructors

$$\begin{aligned}&\texttt{head} : \Pi m,n : \mathbb{N}.(m < n) \to \texttt{string}\ m\ n \to \mathcal{T}_m\\ &\texttt{tail} : \Pi m,n : \mathbb{N}.(m < n) \to \texttt{string}\ m\ n \to \texttt{string}\ m+1\ n\end{aligned}$$

Using a standard technical trick in Coq CITE, we can prove that $\langle\rangle$ is the only object of type $\texttt{string}\ n\ n$, and every object Δ in $\texttt{string}\ m\ n$ has the form $\texttt{head}\ \Delta :: \texttt{tail}\ \Delta$ if $m < n$.

We define the operations Π and Λ that operate on strings and terms, and the operation of *concatenation* that appends a string to a context. The types for these operations are as follows:

- If $\Delta : \texttt{string}\ m\ n$ and $A : \mathcal{T}_n$, then $\Pi\Delta.A : \mathcal{T}_m$.
- If $\Delta : \texttt{string}\ m\ n$ and $A : \mathcal{T}_n$, then $\Lambda\Delta.A : \mathcal{T}_m$.
- If $\Gamma : \mathcal{C}_m$ and $\Delta : \texttt{string}\ m\ n$, then $\Gamma\hat{}\Delta : \mathcal{C}_n$.

These operations are all defined by recursion on the string Δ as follows:

$$\begin{array}{ccc}\Pi\langle\rangle.B \equiv B & \Lambda\langle\rangle.B \equiv B & \Gamma\hat{}\langle\rangle \equiv \Gamma\\ \Pi(A :: \Delta).B \equiv \Pi A(\Pi\Delta.B) & \Lambda(A :: \Delta).B \equiv \lambda A(\Lambda\Delta.B) & \Gamma\hat{}(A :: \Delta) \equiv (\Gamma, A)\hat{}\Delta\end{array}$$

Substitution in Strings. The definition of substitution on strings is very tricky. One natural suggestion would be to define an operation with type

$$\Pi m,n,p : \mathbb{N}.\texttt{string}\ m\ n \to (\mathcal{F}_m \to \mathcal{T}_{m+p}) \to \texttt{string}\ (m+p)\ (n+p)\ .$$

However, as mentioned above, it proved necessary to find a definition that would not involve addition within any type.

After many such false starts, the following solution was arrived at. We define a relation on pairs of natural numbers, which we call *matching*:

$$\langle m,n\rangle \sim \langle p,q\rangle$$

(read: "$\langle m,n\rangle$ *matches* $\langle p,q\rangle$"). This relation is equivalent to $m \leq n \wedge n - m = q - p$. If $\langle m,n\rangle \sim \langle p,q\rangle$, then it is possible to apply a substitution $\mathcal{F}_m \to \mathcal{T}_p$ to a string in $\texttt{string}\ m\ n$ to obtain a string of type $\texttt{string}\ p\ q$.

(We actually place the type $\langle m,n\rangle \sim \langle p,q\rangle$ in Set, as we shall need to define substitution by recursion on the proof that $\langle m,n\rangle \sim \langle p,q\rangle$.)

Definition 3. *Define the relation $\langle m,n\rangle \sim \langle p,q\rangle$ inductively as follows:*

$$\frac{m : \mathbb{N} \quad p : \mathbb{N}}{\langle m,m\rangle \sim \langle p,p\rangle} \qquad \frac{\langle m+1,n\rangle \sim \langle p+1,q\rangle}{\langle m,n\rangle \sim \langle p,q\rangle}$$

We can now define the substitution operation on strings:

Definition 4. *Suppose $\langle m,n\rangle \sim \langle p,q\rangle$. We define, for each $\Delta : \texttt{string}\ m\ n$ and $\rho : \mathcal{F}_m \to \mathcal{T}_p$, the string $\Delta[\rho] : \texttt{string}\ p\ q$, by recursion on the proof of $\langle m,n\rangle \sim \langle p,q\rangle$ as follows:*

– *The base case is* $\langle m,m\rangle \sim \langle p,p\rangle$. *For* $\Delta : \texttt{string}\ m\ m$ *and* $\sigma : \mathcal{F}_m \to \mathcal{T}_p$,

$$\Delta[\sigma] \equiv \langle\rangle : \texttt{string}\ p\ p$$

– *Suppose* $\langle m,n\rangle \sim \langle p,q\rangle$ *was deduced from* $\langle m+1,n\rangle \sim \langle p+1,q\rangle$. *For* $\Delta : \texttt{string}\ m\ n$ *and* $\sigma : \mathcal{F}_m \to \mathcal{T}_p$,

$$\Delta[\sigma] \equiv (\texttt{head}\ \Delta)[\sigma] :: (\texttt{tail}\ \Delta)\begin{bmatrix} \bot \mapsto \bot \\ \uparrow x \mapsto \sigma(x)\{\uparrow\} \end{bmatrix} : \texttt{string}\ p\ q$$

Proof of Strengthening. Now that these operations are in place, it is a straightforward, albeit lengthy, task to formalise van Bentham Jutting's proof of Strengthening.

The proof involves the partitioning of the terms of a PTS into two sets T_{V} and T_{S}. We represent these in the formalization by two functions

$$\texttt{varlike} : \Pi n : \mathbb{N}.\mathcal{T}_n \to \texttt{bool}$$
$$\texttt{sortlike} : \Pi n : \mathbb{N}.\mathcal{T}_n \to \texttt{bool}$$

$\texttt{varlike}\ M$ being $\texttt{true}$ if $M \in T_{\text{V}}$ and $\texttt{false}$ otherwise, and $\texttt{sortlike}$ representing T_{S} in the same way.

The proof also involves a set of sorts $\Sigma(\Gamma, M)$ for every context Γ and term M. We represent this in the formalization by a function

$$\Sigma : \Pi n : \mathbb{N}.\mathcal{C}_n \to \mathcal{T}_n \to \mathcal{S} \to \texttt{Prop}\ .$$

Neither definition presents any problems.

We can proceed to prove the results of van Bentham Jutting's paper one by one. None of the results requires a significant change, and the proof of each follows the proof given in the paper closely.

We note in passing that the only results that require the use of the strings that we went to such trouble to define are these:

Lemma 5 (Typing Lemma).
If $a \in T_{\text{S}}$ *and* $\Gamma \vdash a : A$ *and* $\Gamma \vdash a : B$, *then* $\exists s_1, s_2, C_1, \ldots, C_n[A \twoheadrightarrow \Pi x_1 : C_1.\cdots.\Pi x_n : C_n.s_1 \wedge B \twoheadrightarrow \Pi x_1 : C_1.\cdots.\Pi x_n : C_n.s_2]$ *(*$n \geq 0$*).*

In our formalization, we found that the following weaker result was all that was required for the proof of Strengthening:

$$\forall \Gamma, M, A.\texttt{sortlike}\ M \to \Gamma \vdash M : A \to \exists \Delta, s.A \twoheadrightarrow \Pi\Delta.s$$

Theorem 4. *If* $a \in T_{\text{S}}$, $\Gamma \vdash a : A$ *and* $A \twoheadrightarrow \Pi x_1 : A_1.\cdots.\Pi x_n : A_n.s$, *then* $s \in \Sigma(\Gamma, A)$.

In our formalization:

$$\forall \Gamma, a, A, \Delta, s.\texttt{sortlike}\ a \to \Gamma \vdash a : A \to A \twoheadrightarrow \Pi\Delta.s \to \Sigma\ \Gamma\ a\ s\ .$$

the latter of which is only needed to prove

Corollary 1. *If $A \in T_S$, $\Gamma \vdash A : s$, then $s \in \Sigma(\Gamma, A)$.*

In our formalization:

$$\forall \Gamma, a, s . \texttt{sortlike}\ a \to \Gamma \vdash a : s \to \Sigma\ \Gamma\ a\ s\ .$$

The proof of Strengthening in van Bentham Jutting's paper is concluded by proving the following lemma:

Lemma. If $\Gamma_1, x : A, \Gamma_2 \vdash b : B$ and $x \notin FV(\Gamma_2) \cup FV(b)$, then $\exists B'(B \twoheadrightarrow B' \wedge \Gamma_1, \Gamma_2 \vdash b : B')$.

Theorem. If $\Gamma_1, x : A, \Gamma_2 \vdash b : B$ and $x \notin FV(\Gamma_2) \cup FV(b)$, then $\Gamma_1, \Gamma_2 \vdash b : B$.

We instead prove the following big-step versions:

Lemma 6. *If $\Gamma \vdash b\{\rho\} : B$, Δ is valid, ρ is injective, and $\Delta \subseteq_\rho \Gamma$, then there exists B' such that $B \twoheadrightarrow B'\{\rho\}$ and $\Delta \vdash b : B'$.*

Theorem 5. *If $\Gamma \vdash b\{\rho\} : B$, Δ is valid, ρ is injective, and $\Delta \subseteq_\rho \Gamma$, then there exists B' such that $B \equiv B'\{\rho\}$ and $\Delta \vdash b : B'$.*

Corollary 2. *If $\Gamma \vdash b\{\rho\} : B\{\rho\}$, Δ is valid, ρ is injective, and $\Delta \subseteq_\rho \Gamma$, then $\Delta \vdash b : B$.*

5 Related Work

McKinna and Pollack [2] produced a large formalization of many results in the theory of PTSs, based on a representation of syntax that uses named variables. They use two separate types: V for the bound variables, and P for the free variables, or *parameters.*

From our point of view, this formalization has two principal disadvantages. Firstly, we are often dealing with operations renaming variables, or replacing a variable with a parameter or vice versa; we would prefer not to have to deal with these technicalities. Secondly, we have objects that do not correspond to any term in the syntax — namely, those in which objects of type V occur free. We need frequently to check that every object of type V is bound in the terms with which we are dealing — that they are *closed*, in McKinna and Pollack's terminology.

If one has a reason to use named variables — if one wishes to stay as close as possible to an implementation, if one wishes to check proofs about α-conversion, or if one simply wishes, in McKinna and Pollack's phrase, to "take symbols seriously" — then this is a very good formalization to look at. However, if one's sole concern is to check a proof of a metatheoretic result, then a more abstract representation of syntax simplifies matters greatly.

Barras has produced a formalization in Coq, entitled "Coq in Coq" [3, 10], of the metatheory of the Calculus of Constructions, representing the terms with de Bruijn notation, using Coq's natural numbers for the free and bound variables.

This formalization has the advantages that α-convertible terms are now provably equal, and that there are no objects that do not represent well-formed terms.

However, this formalization is still more complex than our own. It involves operations $\uparrow_k^n$, which raises every de Bruijn index after the kth by n places. We are frequently using lemmas about how these operations interact with each other, or with substitutions. Proofs often proceed by several case analyses involving comparisons of natural numbers, or arithmetical manipulation of the sub- and super-scripts of these operators.

These are the two major formal developments of metatheory of which the author is aware. Also worthy of mention is higher-order abstract syntax [11], which embeds the object theory one is studying within the type theory in which one is working, then takes advantage of the type theory's binding and substitution operations. This did not seem suitable for our purposes; the author thought it important to maintain the separation between object theory and metatheory for this work.

Several other approaches to the representation of syntax have been developed relatively recently, including Gabbay and Pitts' work [12], which develops variable binding as an operation in FM-set theory, and CINNI [13], a formal system which contains both named variable syntax and de Bruijn notation as subsystems. It would be very interesting to see a similar formal development of the metatheory of some system based on either of these.

6 Conclusion

We have formalised in Coq the proofs of many results in the metatheory of PTSs, using a slight variation of the nested data type representation of terms with binding. In the author's opinion, the result is a very simple and elegant formalization. While some thought is often needed to find the correct form for definitions and theorems, the proofs are then short and direct, certainly in the early stages of the metatheory.

When we turn to the proof of Strengthening, this claim can no longer be made: the formalization does then become technically complex. It is difficult to compare this proof with other formalizations: McKinna and Pollack [2] use a different proof of Strengthening, and Barras [3] does not prove Strengthening at all. It is possible that the proof of Strengthening would be just as complex in other formalizations; or that some way of simplifying it in this formalization could be found.

The nested data type representation of syntax is the one the author prefers if one's only interest in formalization is to obtain a guarantee of the correctness of a theorem. It is certainly worthy of being another tool which someone working in formal metatheory should be aware of, and consider using.

Lastly, we may speculate that there may be some theoretical interest in the form the metatheory takes when the nested data type representation of syntax is used. The substitution functions with which we have been dealing are precisely the arrows in the standard way of assigning categorical semantics to a type theory. Possibly the monadic structure of the syntax representation has some implications in this area? The correspondence between the subcontext relation and the satisfaction relation, and likewise between the Weakening Lemma and the Substitution Lemma, is striking. It suggests something along the lines of a fibred category, the base category using renaming operations as arrows, and the fibred category using substitution functios. It remains to be seen whether it is profitable to explore these ideas.

References

1. Adams, R.: Pure type systems with judgemental equality. (Submitted for publication in the Journal of Functional Programming)
2. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. J. Autom. Reasoning **23** (1999) 373–409
3. Barras, B.: Auto-validation d'un sysème de preuves avec familles inductives. PhD thesis, Université Paris 7 (1999)
4. Aczel, P.: Nested data types in constructive type theory. (Talk given at Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy)
5. Barendregt, H.: Lambda calculi with types. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Volume II. Oxford University Press (1992)
6. van Benthem Jutting, L.S.: Typing in pure type systems. Information and Computation **105** (1993) 30–41
7. Bellegarde, F., Hook, J.: Subsitution: A formal methods case study using monads and transformations. Sci. Comput. Program. **23** (1994) 287–311
8. Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. In: CSL '99. Volume 1683 of LNCS., Springer-Verlag (1999) 453–468
9. Bird, R.S., Paterson, R.: de Bruijn notation as a nested datatype. J. Functional Programming **9** (1999) 77–91
10. Barras, B.: A formalization of the calculus of constructions. (Web page) `http://coq.inria.fr/contribs/coq-in-coq.html`.
11. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, ACM Press (1988) 199–208
12. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing **13** (2002) 341–363
13. Stehr, M.O.: CINNI: A generic calculus of explicit substitutions and its application to lambda-, sigma- and pi-calculi. In: Third International Workshop on Rewriting Logic and its Applications (WRLA'2000), Kanazawa, Japan, September 18 – 20, 2000. Volume 36 of Electronic Notes in Theoretical Computer Science., Elsevier (2000)