Formalized metatheory with terms represented by an indexed family of types

Formalized Metatheory with Terms Represented by an Indexed Family of Types

Robin Adams

Royal Holloway, University of London
robin@cs.rhul.ac.uk

Abstract. It is possible to represent the terms of a syntax with binding constructors by a family of types, indexed by the free variables that may occur. This approach has been used several times for the study of syntax and substitution, but never for the formalization of the metatheory of a typing system. We describe a recent formalization of the metatheory of Pure Type Systems in Coq as an example of such a formalization. In general, careful thought is required as to how each definition and theorem should be stated, usually in an unfamiliar 'big-step' form; but, once the correct form has been found, the proofs are very elegant and direct.

1 Introduction

In [1], Bellegarde and Hook show how the terms of a language with binding constructors can be represented as a *nested datatype* — a type constructor that takes types (including possibly its own values) as arguments. This idea has since been used several times for the study of the syntax of such languages, for example in Altenkirch and Reus [2], and Bird and Paterson [3]. However, to the best of the author's knowledge, it has never been used in a formalization of the metatheory of a formal system.

We present here a formalization in Coq of the metatheory of Pure Type Systems (PTSs) using this representation for the set of terms. We prove all of the results about arbitrary PTSs given in Barendregt [4], including Subject Reduction and Uniqueness of Types for functional PTSs. The formalization also includes van Bentham Jutting's proof of Strengthening [5].

There have been several formalizations of the metatheory of formal systems in the past, two of the largest being McKinna and Pollack [6] and Barras [7]; and so we shall be able to compare the strengths and weaknesses of this approach with those of the previous.

The indexed family approach proves to have quite limited expressive power. We cannot define all the operations nor state all the results in the form we are used to. Careful thought was often needed as to what form a definition or theorem could take. In general, it was found that operations involving all variables simultaneously were easy to represent in this formalization, while those involving single variables were difficult to represent. For example, we can define the operation of substituting for every variable simultaneously, but not that of

J.-C. Filliâtre et al. (Eds.): TYPES 2004, LNCS 3839, pp. 1–16, 2006.

substituting a given term for an arbitrary single variable. The author found himself calling operations of the first kind 'big-step' operations, and those of the second kind 'small-step'. These are as good names as any, and we shall continue to use them.

To set against this, it was found that, once the correct form for each definition and theorem had been arrived at, the proofs themselves were simple, short, elegant and easily constructed. We find ourselves spending much less time on technical details than in most formalizations of metatheoretic work. Perhaps most importantly, those technicalities that we do have to deal with have a more type-theoretic flavour: the objects with which one deals are those one would expect to find in the metatheory of a type theory, rather than low-level aspects of the mechanisms of the formalization.

In particular, each specific instance of a small-step operation we need (such as the substitution involved in β-reduction) is definable as a special case of a big-step operation. The results that we need about it are likewise derivable as special cases of results about the big-step operations.

There were two maxims that, at several points in this work, it proved wise to follow, and which would seem to be more widely applicable to the formalization of mathematics in general. They can be stated briefly as: *favour recursive definitions over inductive ones*, and *avoid arithmetic in types*. We discuss them in Section 2. We proceed in Section 3 to a description of the formalization itself, showing how the two maxims are applied, and showing the big-step form of each definition and theorem.

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

2 Maxims for Formalization

There were three general principles that it proved wise to follow in this work. One — that 'big-step' definitions should be preferred to 'small-step' — is peculiar to this work, and we shall discuss it in Section 3. We wish here to discuss the other two maxims, which should be more generally applicable to other formalizations.

2.1 Recursive Definitions Versus Inductive Families

Our first maxim is:

> When defining a family of types $F : I \to \texttt{Set}$ over an inductive type I, if possible define F by recursion over I, rather than as an inductive family

or more briefly, *favour recursive definitions over inductive ones*.

The difficulties of using inductive families are well known among the Coq community. The standard example is the family $\mathcal{V}_n$ of *vectors* of length n (over a given type A). We can define this either as an inductive family, with constructors:

$$\frac{}{\langle\rangle : \mathcal{V}_0} \qquad \frac{a : A \quad v : \mathcal{V}_n}{(a :: v) : \mathcal{V}_{n+1}}$$

or by recursion on the index n, as follows:

$$\mathcal{V}_0 = \mathbf{1} \qquad \mathcal{V}_{n+1} = A \times \mathcal{V}_n$$

(Here, $\mathbf{1}$ is the type with a single canonical object, called `unit` in the Coq library, and $\times$ of course denotes the non-dependent product of two types.)

The practical difference between these two definitions lies in how easy it is to deduce information about an object of type $\mathcal{V}_t$ from the shape of the term $t : \mathbb{N}$. In particular, we frequently want to use the fact that a term $s : \mathcal{V}_0$ must be equal to $\langle\rangle$, and that a term $s : \mathcal{V}_{t+1}$ must have the form $a :: v$ for some $a : A$ and $v : \mathcal{V}_t$. This can be deduced immediately if we are using the recursive definition, but not with the inductive definition.

There is a standard technical trick for overcoming this difficulty that is known as folklore among the Coq community. It appears unattributed in Letouzey's message [8]. To the author's knowledge, this technique has never before been put into print, and Letouzey may well be its inventor.

It can be summarised as follows:

- Define a function which should be the identity function on the inductive family.
- Prove that the function is indeed the identity function.
- Deduce from this fact theorems allowing case analysis on the objects of the inductive family.

In the case of $\mathcal{V}_n$, we provide ourselves with the destructors $\texttt{head} : \Pi n : \mathbb{N}.\mathcal{V}_{n+1} \to A$ and $\texttt{tail} : \Pi n : \mathbb{N}.\mathcal{V}_{n+1} \to \mathcal{V}_n$:

$$\texttt{head}\ n\ (a :: v) = a \qquad \texttt{tail}\ n\ (a :: v) = v$$

(These are two instances where case analysis on $\mathcal{V}_n$ can be applied successfully.) We can then define our 'pseudo-identity' function $\texttt{pseudoid} : \Pi n : \mathbb{N}.\mathcal{V}_n \to \mathcal{V}_n$ by recursion *on* n as follows:

$$\begin{aligned} \texttt{pseudoid}\ 0\ v &= \langle\rangle && (v : \mathcal{V}_0) \\ \texttt{pseudoid}\ n+1\ v &= (\texttt{head}\ n\ v) :: (\texttt{tail}\ n\ v) && (v : \mathcal{V}_{n+1}) \end{aligned}$$

We can prove the following result by induction *on* v:

$$(\forall n : \mathbb{N})(\forall v : \mathcal{V}_n)\texttt{pseudoid}\ n\ v = v \ . \tag{1}$$

From this, the following case analysis theorems can be easily deduced:

$$(\forall v : \mathcal{V}_0)\, v = \langle\rangle \tag{2}$$

$$(\forall n : \mathbb{N})\, (\forall v : \mathcal{V}_{n+1})\, v = (\texttt{head}\ n\ v) :: (\texttt{tail}\ n\ v) \tag{3}$$

One way of viewing this construction is as *building a bijection between the inductively defined family and the recursively defined family.* More precisely, we have built a bijection between $\mathcal{V}_0$ and $\mathbf{1}$, and between $\mathcal{V}_{n+1}$ and $A \times \mathcal{V}_n$. The

constructors form one half of the bijection: the 'cons' constructor :: is a (Curried) mapping $A \times \mathcal{V}_n \to \mathcal{V}_{n+1}$, and $\langle\rangle$ can be seen as a mapping $\mathbf{1} \to \mathcal{V}_0$. Our destructors form the other half: `head` and `tail` together give a mapping $\mathcal{V}_{n+1} \to A \times \mathcal{V}_n$, and we take the trivial mapping $\mathcal{V}_0 \to \mathbf{1}$.

The function `pseudoid` is the composition of these two halves: `pseudoid` 0 is the composition

$$\mathcal{V}_0 \to \mathbf{1} \xrightarrow{\langle\rangle} \mathcal{V}_0 \ ,$$

and `pseudoid` $n+1$ is the composition

$$\mathcal{V}_{n+1} \xrightarrow{\langle \texttt{head}, \texttt{tail} \rangle} A \times \mathcal{V}_n \xrightarrow{::} \mathcal{V}_{n+1} \ .$$

Our theorem (1) then verifies that these compositions do produce the identity mapping; that is, that `cons` and `emp` are left inverses to $\langle \texttt{head}, \texttt{tail} \rangle$ and the trivial mapping, respectively. (That they are also right inverses is immediate from the definitions of `head` and `tail`.) In theorems (2) and (3), we then use the fact that case analysis is possible on $\mathbf{1}$ and $A \times \mathcal{V}_n$ to prove it is possible on $\mathcal{V}_0$ and $\mathcal{V}_{n+1}$.

Conversely, we can see intuitively that any way of performing case analysis on the elements of $\mathcal{V}_n$ would yield appropriate functions $\mathcal{V}_0 \to \mathbf{1}$ and $\mathcal{V}_{n+1} \to A \times \mathcal{V}_n$. So we make the following, as yet rather imprecise, conjecture:

> Case analysis is possible on the objects of an inductively defined family of types if and only if the family is isomorphic to a recursively defined family.

2.2 Arithmetic Within Types

Our second maxim is:

> When using a family of types indexed by `nat`, make sure that the index term never involves `plus` or `times`

or, more briefly, *avoid arithmetic within types.*

Let us continue with our example of the family $\mathcal{V}_n$ of types of vectors that we have introduced. A natural operation to define is the action of *appending* two vectors:

$$\frac{u : \mathcal{V}_m \quad v : \mathcal{V}_n}{u \hat{} v : \mathcal{V}_{m+n}}$$

$$\langle\rangle \hat{} v = v, \qquad (a :: u) \hat{} v = a :: (u \hat{} v)$$

We now try to prove that this operation is associative:

$$u \hat{} (v \hat{} w) = (u \hat{} v) \hat{} w$$

We have a problem. The left-hand side of this equation has type $\mathcal{V}_{m+(n+p)}$, while the right-hand side has type $\mathcal{V}_{(m+n)+p}$. These two types, while provably equal, are not convertible, so the proposition as we have written it is not well-formed.

We are trying to treat two terms with different types as if they had the same type.

We can also meet the converse problem, where we require one term to have two different types within the same formula; say type $\mathcal{V}_{m+Sn}$ at one point and type $\mathcal{V}_{Sm+n}$ at another.

Should these two maxims ever conflict, the second maxim should take priority over the first. The first involves a known quantity of work: once the case analysis lemmas have been proven by the known method described above, then we can forget about the fact the family was defined inductively. Our two maxims conflicted at only one point in this formalization: the definition of the family of types of *strings*. We shall say more about this in Section 3.3.

3 Description of the Formalization

We proceed to a description of the formalization of the metatheory of PTSs. Most of the formalization shall not be described in any detail; we shall concentrate instead on the parts that are novel to this formalization, particularly those definitions and theorems which need to be stated in an unfamiliar form owing to the fact that we are working with the indexed family representation of terms. A detailed description of the formalization can be found at [9].

We have already discussed two themes that occurred several times in this work: the preference for recursive definitions of families of types over inductive ones, and the need to avoid arithmetic within types. A third is that, when terms are defined as an indexed family, it is easier to define operations and prove theorems that deal with all variables simultaneously, than those that deal with a single variable. For example, it is easier to define the operation of substituting for every variable simultaneously, than that of substituting for a single given variable. We shall give the name of 'big-step' operations and theorems to the first kind, and 'small-step' to the second.

Our definitions of replacement (substitution of variables for variables), substitution, and the subcontext relation all get big-step definitions, and we also find ourselves needing a *satisfaction* relation, which can be seen as a big-step version of the typing relation. We found it easiest to base the definition of reduction and conversion on parallel reduction, rather than one-step reduction; this again could possibly be seen as a big-step/small-step distinction. (In this choice, we follow McKinna and Pollack [6].) The Weakening, Substitution, Subject Reduction and Context Conversion results were all found to be easier to state and prove in a big-step form. The small-step instances of these operations, relations and results that we later need can all be derived as special cases of the big-step forms.

3.1 Grammar

Terms as a Nested Datatype. The idea of representing terms as a nested datatype first appeared in Bellegarde and Hook [1]. 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 [2] or Bird and Paterson [3]; the latter 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 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, suppose we wish to represent the terms of the untyped lambda calculus. Let us assume we have, for every type X, a type $X_\perp$ consisting of a copy $\uparrow x$ of each $x : X$, together with one extra object, $\perp$. (In Coq, this would be the type `option` X provided by the Coq library.) We can then represent the terms of the untyped lambda calculus by the inductive family $\mathcal{T}_V$ whose constructors are as follows.

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

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 $\sigma(u) : \mathcal{T}_V$. The definition that we would naturally write down is as follows.

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

(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 directly the special case where σ 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:

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

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

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

Altenkirch and Reus [2] show how the type operator $\mathcal{T}$, the constructor `var`, and the substitution operation [ ] form a *Kleisli triple*, a concept closely related to that of a monad.

Bird and Paterson [3] 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.

Terms as an Indexed Family. For our 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 these as "`nat`-indexed terms".

We define a family of finite types $\mathcal{F}_n$ ($n : \mathbb{N}$), $\mathcal{F}_n$ having exactly n distinct canonical objects. Following our first maxim (see Section 2), we define $\mathcal{F}$, not as an inductive family of types, but as a recursive function thus:

$$\mathcal{F}_0 = \emptyset, \qquad \mathcal{F}_{n+1} = (\mathcal{F}_n)_\bot \ .$$

Here, $\emptyset$ (`empty`) is the empty type, and $X_\bot$ (`option` X) is a type consisting of a copy $\uparrow x$ of each object x of X together with one extra object $\bot$. Both of these types are provided by the Coq library.

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

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

We proceed to define the replacement and substitution operations, as discussed in the previous section. We can then prove the following various forms of the Substitution Lemma:

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

Reduction Relation. 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:

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

$$\frac{A \rhd A' \quad B \rhd B'}{\Pi A B \rhd \Pi A' B'} \qquad \frac{A \rhd A' \quad M \rhd M'}{\lambda A M \rhd \lambda A' M'} \qquad \frac{M \rhd M' \quad N \rhd N'}{(\lambda A M) N \rhd M' \left[\begin{smallmatrix} \bot \mapsto N' \\ \uparrow x \mapsto x \end{smallmatrix}\right]}$$

Note, in particular, the way that substitution of N' for $\bot$ is defined in terms of our big-step substitution in the final clause.

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 prove that $\rhd$ satisfies the diamond condition, and deduce the Church-Rosser Theorem.

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

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

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 shall write $\Gamma(x)$ for `typeof` _ x Γ in this paper. 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):

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

3.2 Typing Relation

We declare the axioms $\mathcal{A} : \mathcal{S} \to \mathcal{S} \to$ `Prop` and rules $\mathcal{R} : \mathcal{S} \to \mathcal{S} \to \mathcal{S} \to$ `Prop` of the arbitrary PTS with which we are working as parameters, and we are then 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 Figure 1).

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

When we use named variables, the subcontext relation is defined as follows:

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

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

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

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

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

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

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

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

> 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. 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 $x : A$.

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. In fact, we have met these functions before: they are the *renaming* functions $\rho : \mathcal{F}_m \to \mathcal{F}_n$ used by the replacement operation.

We 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:

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 result becomes:

Theorem 1 (Weakening). *If $\Gamma \subseteq_\rho \Delta$, $\Gamma \vdash M : A$, and Δ is valid, then $\Delta \vdash M\{\rho\} : A\{\rho\}$.*

This is a big-step version of the Weakening property: the small-step version would be

If $\Gamma, \Delta \vdash M : A$ and $\Gamma \vdash B : s$ then $\Gamma, x : B, \Delta \vdash M : A$.

Apart from the definition of the subcontext relation, the big-step version of the Weakening Lemma is not particularly novel: it appears in many informal and formal developments, including McKinna and Pollack [6]. The form that the next result takes is probably more surprising:

Substitution. What form should the Substitution result 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

$$\Gamma \vdash \sigma(x_1) : A_1[\sigma],\ \Gamma \vdash \sigma(x_2) : A_2[\sigma],\ \cdots,\ \Gamma \vdash \sigma(x_m) : A_m[\sigma]$$

then $\Gamma \vdash M[\sigma] : B[\sigma]$.

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

It is then clear how the result should read with `nat`-indexed variables. Let us first introduce a definition to abbreviate the hypotheses.

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 version of the Substitution property reads:

Theorem 2 (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, $\rhd$, *on* $\mathcal{C}_n$ *as follows:* $\Gamma \rhd \Delta$ *iff* $(\forall x : \mathcal{F}_n)\Gamma(x) \rhd \Delta(x)$.

We can now prove:

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

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

Other Results. The formalization also contains proofs of Context Conversion, the Generation lemmas, Type Validity (that if $\Gamma \vdash M : A$ then either A is a sort or $\Gamma \vdash A : s$ for some sort s), Predicate Reduction and the Uniqueness of Types result for functional PTSs. Apart from Context Conversion, which takes the expected big-step form, these results take the same form, and the proofs follow the same lines, as the paper development.

3.3 Strings of Binders

The proof of the Strengthening Theorem in van Bentham Jutting's paper [5] 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 formalize when using `nat`-indexed terms is the use of terms of the form $\Lambda\Delta.M$ 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 [5] 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 $\Gamma\hat{}\Delta$. 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 we have discussed, it was found to be important to *avoid arithmetic within types* at all costs. This proved impossible to do without violating our first maxim, to prefer recursive definitions to inductive ones. Even then, it was difficult to find a set of definitions that avoids the need for addition, particularly within the type of the substitution operation.. 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 `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 $A_m : \mathcal{T}_m$, $A_{m+1} : \mathcal{T}_{m+1}$, $\ldots$, $A_{n-1} : \mathcal{T}_{n-1}$. If $n = m$, the type.has only one member (the empty string); and if $n > m$, `string` m n is empty.

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

Using the standard technical trick described above in Section 2.1, 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 $A :: \Delta'$ 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:

$$\frac{\Delta : \texttt{string}\ m\ n \quad A : \mathcal{T}_n}{\Pi\Delta.A : \mathcal{T}_m} \qquad \frac{\Delta : \texttt{string}\ m\ n \quad A : \mathcal{T}_n}{\Lambda\Delta.A : \mathcal{T}_m} \qquad \frac{\Gamma : \mathcal{C}_m \quad \Delta : \texttt{string}\ m\ n}{\Gamma\hat{}\Delta : \mathcal{C}_n}$$

These operations can all be defined by recursion on the string Δ.

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

$$n \leq m \wedge m - n = p - q\ .$$

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 $\texttt{Set}$, 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 $\rho : \mathcal{F}_n \to \mathcal{T}_q$, 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* Δ : `string` m n *and* $\sigma : \mathcal{F}_m \to \mathcal{T}_p$,

$$\Delta[\sigma] \equiv (\texttt{head}\ \Delta)[\sigma] :: (\texttt{tail}\ \Delta)[\sigma_\bot] : \texttt{string}\ p\ q$$

Now that these operations are in place, it is a straightforward, albeit lengthy, task to formalise van Bentham Jutting's proof of Strengthening. We omit the details here; we refer the interested reader to [9]. We note in passing only that, as is by now to be expected, the Strengthening Theorem itself takes a big-step form:

Theorem 4 (Strengthening). *If* $\Gamma \subseteq_\rho \Delta$, ρ *is injective,* $\Delta \vdash M : A$, *and* Γ *is valid, then* $\Gamma \vdash M : A$.

4 Systems with Judgemental Equality

The author's original motivation for this work was to check a technically complex proof of a result of his in the theory of PTSs [10], in order to obtain a guarantee of its correctness. It is worth stating briefly how the formalization of the system with judgemental equality proceeds, as the metatheory of these systems have not often been formalized.

We use the same types $\mathcal{T}_n$ of terms and $\mathcal{C}_n$ of contexts that we have already constructed. The system with judgemental equality has two judgement forms:

$$\Gamma \vdash M : A \text{ and } \Gamma \vdash M = N : A\ .$$

We build these using a mutual inductive definition:

```
Inductive PTS' :
  forall n, context n -> term n -> term n -> Prop :=
...
with PTSeq :
  forall n, context n -> term n -> term n -> term n -> Prop :=
...
```

We define the notion of a *valid context* of type $\mathcal{C}_n$, by induction on n:

$$(\langle\rangle \text{ is valid}) \equiv \top$$
$$(\langle\Gamma, A\rangle \text{ is valid}) \equiv \exists s : \mathcal{S}.\Gamma \vdash A : s$$

We also need the notion of *equality of contexts*: we define the proposition $\Gamma = \Delta$ for Γ and Δ both of the same type $\mathcal{C}_n$.

$$(\langle\rangle = \langle\rangle) \equiv \top$$
$$(\langle\Gamma, A\rangle = \langle\Delta, B\rangle) \equiv \Gamma = \Delta \wedge \exists s : \mathcal{S}.\Gamma \vdash A = B : s$$

The definition of satisfaction is similar to the one we used in PTSs: for $\Gamma : \mathcal{C}_n$, $\Delta : \mathcal{C}_m$, and $\sigma : \mathcal{F}_m \to \mathcal{T}_n$, we define

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

We also need the notion of two substitutions, each of which satisfies Δ under Γ, being equal. For $\Gamma : \mathcal{C}_n$, $\Delta : \mathcal{C}_m$, and $\sigma, \sigma' : \mathcal{F}_m \to \mathcal{T}_n$, we define

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

We then prove:

Theorem 5.

1. ***Context Validity*** *If $\Gamma \vdash J$, then Γ is valid.*
2. ***Context Conversion*** *If $\Gamma \vdash J$, $\Gamma = \Delta$, and Δ is valid, then $\Delta \vdash J$.*
3. ***Substitution*** *If $\Gamma \models \sigma :: \Delta$, $\Delta \vdash J$, and Γ is valid, then $\Gamma \vdash J[\sigma]$.*
4. ***Weakening*** *If $\Gamma \vdash J$, $\Gamma \subseteq_\rho \Delta$, and Δ is valid, then $\Delta \vdash J\{\rho\}$.*
5. ***Functionality*** *If $\Gamma \models \sigma = \sigma' :: \Delta$, $\Delta \vdash M : A$, and Γ is valid, then $\Gamma \vdash M[\sigma] = M[\sigma'] : A[\sigma]$.*

In each case, J stands for either of the judgement bodies $M : A$ or $M = N : A$. In the formalization, each of these statements (except Functionality) is therefore two theorems. They are proven simultaneously — that is, we prove their conjunction by a simultaneous induction on the derivation of the premise.

We note that Weakening is not needed in the proof of Substitution. Weakening can either be proven by an induction on its premise, or as a special case of Substitution.

The Start, Generation and Type Validity lemmas can also be proven; these all take the expected form.

The form that the statement of Subject Reduction takes is:

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

This is a very difficult property to prove for systems with judgemental equality, either on paper or formally. Its proof can be seen as the main result in the paper [10]. The formalization of this proof is not yet complete.

5 Related Work

McKinna and Pollack [6] 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*. It was a concern of theirs not to gloss over such matters as α-conversion, as is usually done in informal develpoments — to 'take symbols seriously', in their words. If one shares this concern, or one wishes to stay close to a particular implementation, then this is an excellent formalization to look at.

However, if one's concern is solely to obtain a guarantee of the correctness of a metatheoretic result, 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.

Barras produced a formalization in Coq of the metatheory of the Calculus of Constructions, , entitled "Coq in Coq" [7, 11], in the hope of certifying Coq itself — or, at least, certifying a proof checker that is as close to Coq as Gödel's Theorem allows. As terms are represented internally in Coq with de Bruijn notation, so does Barras's formalization, using Coq's natural numbers for the free and bound variables.

Again, this formalization quickly becomes technically complex. It involves operations such as $\uparrow_k^n$, 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.

Also worthy of mention is higher-order abstract syntax [12], 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 more recently, including Gabbay and Pitts' work [13], which develops variable binding as an operation in FM-set theory, and CINNI [14], 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

Which formalization of a given piece of mathematics one prefers depends, of course, on what one intends to use the formalization for, and simply on personal taste. If one's sole concern is to obtain a guarantee of the correctness of the proof of a metatheoretic result, then one has free choice of which representation of terms one uses. We have seen that, if one chooses this indexed family representation, then careful thought is needed over the form of definitions and theorems; but, once the correct forms have been found, the proofs are short, simple, direct and elegant.

The technicalities in our formalizations mostly involve the replacement and substitution functions, how they interact with each other and with reduction, conversion and so forth. These are quite natural objects to find in the metatheory of a type theory, so we feel this work has a more 'type-theoretic' flavour than is the case with many other formalizations.

One's time is spent thinking about the most convenient form for definitions and theorems, rather than proving many technical lemmas. This suits the author's preferences nicely; others may prefer an approach that, for example, lends

itself better to automation of the technical parts. Nevertheless, it shall hopefully be useful to see what shape formalized metatheory takes when the indexed family representation of terms is used.

References

1. Bellegarde, F., Hook, J.: Subsitution: A formal methods case study using monads and transformations. Sci. Comput. Program. **23** (1994) 287–311
2. 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
3. Bird, R.S., Paterson, R.: de Bruijn notation as a nested datatype. J. Functional Programming **9** (1999) 77–91
4. 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)
5. van Benthem Jutting, L.S.: Typing in pure type systems. Information and Computation **105** (1993) 30–41
6. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. J. Autom. Reasoning **23** (1999) 373–409
7. Barras, B.: Auto-validation d'un sysème de preuves avec familles inductives. PhD thesis, Université Paris 7 (1999)
8. Letouzey, P.: Vectors. Message to Coq-Club mailing list (2004) `http://pauillac.inria.fr/pipermail/coq-club/2004/001265.html`.
9. Adams, R.: A formalization of the theory of PTSs in Coq. Available online at `http://www.cs.rhul.ac.uk/~robin/coqPTS/docu.dvi` (2005)
10. Adams, R.: Pure type systems with judgemental equality. (Accepted for publication in the Journal of Functional Programming)
11. Barras, B.: A formalization of the calculus of constructions. (Web page) `http://coq.inria.fr/contribs/coq-in-coq.html`
12. 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
13. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing **13** (2002) 341–363
14. 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)