On the Formal Metatheory of the Pure Type Systems using One-sorted Variable Names and Multiple Substitutions*

Sebastián Urciuoli
Universidad ORT Uruguay, Uruguay
urciuoli@ort.edu.uy

We develop formal theories of conversion for Church-style lambda-terms with Pi-types in first-order syntax using one-sorted variables names and Stoughton's multiple substitutions. We then formalize the Pure Type Systems along some fundamental metatheoretic properties: weakening, syntactic validity, closure under alpha-conversion and substitution. The whole development has been machine-checked using the Agda system. We compare our formalization with others related. Our work demonstrates that the mechanization of dependent type theory by using conventional syntax and without identifying alpha-convertible lambda-terms is feasible.

1 Introduction

In [26], Tasistro et al. developed a framework in Agda [22] containing Stoughton's theories of substitution and α-conversion [25] for the pure λ-calculus in its conventional syntax, i.e., first-order with only one sort of names to serve for both free and bound variables. Their prime motivation was to test whether such concrete approach was in any way amenable to full formalization.

The use of Stoughton's simultaneous substitutions brings about the possibility to define a capture-avoiding and structurally recursive substitution operation by renaming the bound variables at the same time the operation is taking place. In contrast, the classical formal definition for the unary substitution, e.g., by Curry-Feys [11] and Hindley-Seldin [15], is non-structural because, in the case of λ-abstractions, it invokes itself twice, once to rename the bound name, and again to perform the actual substitution; the latter is on a λ-term that it is not a proper component of the input. Well-founded recursion is not ideal in mechanizations because, in general, they require one to conduct many proofs by complete induction on the length of the syntax. In addition to being structurally recursive, Stoughton's substitution operation actually renames all bound variables without exempting those who may actually do not cause trouble. Perhaps counterintuitively, this turns out to be very welcome because many proofs that follow the structure of the syntax can be carried out smoothly without having to scrutinize the name of the bound variables in the case of λ-abstractions, leading to a reduction in the number of cases considered.

The framework has been put to the test since then to verify many results about the pure λ-calculus and the simply-typed λ-calculus (STLC). In [9], the authors formalized the Church-Rosser (CR) theorem for the pure λ-calculus with β-conversion by the argument of Martin-Löf and Tait and subject reduction for the STLC. In [10], the authors mechanized a proof of the standardization theorem by Kashima [17]. In [31], a formal proof of the strong normalization theorem for STLC by Joachimski and Matthes [16] was presented. Finally, in [30] a proof of strong normalization for System T by Girard [12] was mechanically verified. In spite of having to consider α-conversion explicitly, the reports revealed that the workload in terms of labour was still manageable.

*This work is partly supported by Agencia Nacional de Investigación e Innovación (ANII), Uruguay

Chaudhuri, Nantes-Sobrinho (Eds.): International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2025)
EPTCS ??, 2025, pp. 1–17, doi:10.4204/EPTCS.??.??

Now in this work we wish to test whether our approach scales well to more complex languages and so we formalize a λ-calculi with dependent types: the Pure Type Systems (PTS) [3]. The PTS is a generalization of the λ-cube that represent many systems under the same formalism and so it allows to study many properties for all of them.

In order to mechanize the methatheory of the PTS, first we have to extend the syntax of the framework with type-annotated or Church-style λ-abstractions and Π-types. As a consequence, we have to revisit an important aspect of the theory of substitutions, namely that of restrictions, i.e., the confinement of their domains, due to some technical reason that we will address in due course. Also, we generalize the type of variables so it can be any in one-to-one correspondence with the natural numbers; in previous works, names were identified with numbers, which takes away a bit of the fun of using names. Besides, with this generalization we allow to use strings for the variables thus narrowing the gap between a potential practical implementation of a type-checker (using names) and its certification. Finally, we give a more accurate definition of α-conversion which will enable us to prove a key result in the theory of α-conversion using simpler methods than in previous work. All four previous features compelled us to virtually rewrite the framework entirely. The result is an Agda library with theories of substitution and α-conversion for the underlying language of the PTS: a Church-style λ-calculus with Π-types.

Once we have the new framework we formalize some fundamental metatheoric properties of the PTS, including: thinning (weakening), syntactic validity and closure under α-conversion and substitution. Following [20], we define the type system by using generalized induction, a technique which will enable us to prove thinning by using structural induction, and then we will show that such presentation is extensionally equivalent to a more standard one that does not feature such rule schemes. Besides, in the course of the development of the metatheory we discuss a problem we had to face arising from the use of renaming substitutions that only appear in the context of dependent types and which has not been commented anywhere yet to the best of our knowledge.

The structure of this work is as follows. In the next section we introduce the new framework. Some results are adapted from previous work and some others are new. From now on, all results constitute new development. In Section 3 we present the PTS along with some basic properties. In Section 4 we formalize thinning, syntactic validity and closure under α-conversion and substitution. In Section 5 we comment on related work. Finally, in Section 6 we give some code metrics and conclude.

Throughout this paper we will use Agda notation for definitions and lemmas, and a mix with English for the proofs with the hope of making reading more enjoyable. Some background on any dependently-typed programming is preferable, though it might be enough for some readers to have an understanding in functional programming. The complete sources can be found at the following link: `https://github.com/surciuoli/pts-metatheory`.

2 The Framework

In this section we define the syntax of the λ-terms, substitution and α-conversion based on the work by Stoughton [25] (extended to the language of our interest). Also, we prove some important results that we will need in the following sections.

2.1 Syntax

Let $\mathcal{V}$, the *variables*, be any type such that, first, propositional equality is decidable, and second, there are two functions `encode` and `decode` such that `encode` maps every variable to a number and `decode` is its

```
data Λ : Set where
  c : 𝒞 → Λ
  v : 𝒱 → Λ
  λ[_:_]_ : 𝒱 → Λ → Λ → Λ
  Π[_:_]_ : 𝒱 → Λ → Λ → Λ
  _·_ : Λ → Λ → Λ
```

(a) Syntax

```
fv : Λ → List 𝒱
fv (c _) = []
fv (v x) = [ x ]
fv (λ[ x : A ] M) = fv A ++ (fv M - x)
fv (Π[ x : A ] B) = fv A ++ (fv B - x)
fv (M · N) = fv M ++ fv N
```

(b) Free Variables

Figure 1: λ-terms

right inverse. We shall use meta-variables x, y, etc. to range them. Let $\mathscr{C}$, the *constants*, be any type and whose elements are ranged by k and s. The abstract syntax of the λ*-terms* (Λ) is defined inductively in Fig. 1a.

In Fig. 1b we define the function that returns the free names in a given λ-term, where `_++_` is list concatenation and `_-_ : List` $\mathscr{V}\to\mathscr{V}\to$`List` $\mathscr{V}$ the function that deletes every occurrence of some name in a given list. We can then base ourselves upon `fv` to define whether a name occurs free in some λ-term or not by: `x * M = x ∈ fv M` and read x is free in M and oppositely, `x # M = x ∉ fv M` and read x is fresh for M.

The following results about lists will enable us to construct and destruct derivations of $x*M$ and $x\#M$ as if they were inductively defined:

Lemma 2.1. *1.* `appIn : ∀ {x} {xs ys : List 𝒱} → x ∈ xs ++ ys ↔ x ∈ xs ⊎ x ∈ ys`
2. `delIn : ∀ {x y xs} → x ∈ xs - y ↔ x ≢ y × x ∈ xs`
3. `appNotIn : ∀ {x} {xs ys : List 𝒱} → x ∉ xs ++ ys ↔ x ∉ xs × x ∉ ys`
4. `delNotIn : ∀ {x y xs} → x ∉ xs - y ↔ (x ≡ y ⊎ x ∉ xs)`

where `_⊎_` is disjoint union, `_×_` non-dependent sum and `A ↔ B = (A → B) × (B → A)`.

2.2 The Choice Function

To implement Stoughton's substitution operation, first we need to define a function that *chooses* a fresh name for a list of given names. This list will contain the free names in the image of the substitution which need to be avoided when renaming the bound variables.

In previous version of the framework, names have been identified with the natural numbers, so the choice function had type: `χ' : List ℕ → ℕ`. There it was shown that such function actually returns a number fresh for the input list:

Lemma 2.2. `xpfresh : ∀ xs → χ' xs ∉ xs`

Now in this development, the names in the list have an abstract type $\mathscr{V}$, hence we also need a more abstract choice function. To this end, we define a new function X' such that, given a list xs, it encodes xs, then calls the previous function and finally decodes the result. Formally:

```
X' : List 𝒱 → 𝒱
X' xs = from (χ' (map to xs))
```

It follows that this new function also selects a fresh name for the input list:

Lemma 2.3. `Xpfresh : ∀ xs → X' xs ∉ xs`

Proof. By Lemma 2.2 and by the fact that `decode` is the right inverse of `encode`. □

2.3 Substitutions

Substitutions are functions from variables to terms:

```
Sub = 𝒱 → Λ
```

We shall use the letter σ possibly with primes to range them. We define ι as the *identity* substitution, and an *update* operation on substitutions `_,_:=_ : Sub →𝒱→Λ→Sub` such that for any given substitution σ, λ-term N and names x and y, $(\sigma, x := N)\, y$ yields N if x equals to y and σy otherwise.

To reason about substitutions it turns out convenient to confine their domain to finite subsets of them. In previous work, the authors introduced the *restriction* type, whose objects were just pairs of substitutions and λ-terms (σ, M) and which would be used anywhere it was required to reason about the extension of σ only up to the free names in M. In our case, however, we shall see that we need a more flexible definition, therefore we define restrictions as pairs of substitutions and *list of names*:

```
Res = Sub × List 𝒱
```

For instance, we can use them to extend the choice function to substitutions:

```
X : Res → 𝒱
X (σ , xs) = X' (concat (map (fv ∘ σ) xs))
```

where `concat` is the function that flattens a list of lists.

With our machinery for restrictions we can prove this extended choice function `X` also returns a fresh name. Let the relations of free and fresh variables be extended to restrictions by:

```
x *⌊ (σ , xs) = ∃ λ y → y ∈ xs × x ∈ fv (σ y)
x #⌊ (σ , xs) = ∀ y → y ∈ xs → x # σ y
```

Then we have that $X(\sigma, xs)$ is fresh for every $y \in xs$:

Lemma 2.4. `Xfresh : ∀ σ xs → X (σ , xs) #⌊ (σ , xs)`

Proof. By using lists properties and Lemma 2.3. □

Without further ado, we define by recursion on the syntax the substitution operation, which given any substitution σ and λ-term M, it replaces the free names in M by their corresponding images in σ while renaming the bound variables at the same time to prevent any possible name capture:

```
_•_ : Λ → Sub → Λ
c k • _ = c k
v x • σ = σ x
M · N • σ = (M • σ) · (N • σ)
λ[ x : A ] M • σ = λ[ y : A • σ ](M • σ , x := v y) where y = X (σ , fv M - x)
Π[ x : A ] B • σ = Π[ y : A • σ ](B • σ , x := v y) where y = X (σ , fv B - x)
```

In the equation for λ-abstractions we could have defined `y = X (σ , λ[ x : A ] M)`, and analogously in the case of Π-types, and therefore saved us the trouble of revisiting the definition of restrictions. Nevertheless, we find this to be rather unsatisfactory because in that case we would be excluding from consideration those names that occur free in the image of A, but not necessarily in that of $\mathrm{fv}\, M - x$, and which are safe candidates to rename the bound variable. Hence we would not be choosing the *first* name available but *some* one. We take the view that selecting the first name available is the best solution because it is the most straightforward way to formally specify such operation. Besides, this way we are aligned with the formal treatment of the λ-calculus in the literature [11, 15].

We shall use the next abbreviation for unary substitution: `M [ x := N ] = M • ι , x := N`.

The first result we have concerning the substitution operation is that it is actually capture-avoiding:

Lemma 2.5. `noCapture : ∀ {x M σ} → x ∈ fv (M • σ) ↔ x *⌊ (σ , fv M)`

Read right-to-left: if x is free in the image of σ then it will remain so in the final result. The other direction is important as well, and it establishes that no free names are introduced by the operation other than those occurring in the image of the σ (restricted to the free names in M).

Next we have some other nice properties on substitutions that we will use thoroughly in the next sections. First, let equality on restrictions be defined by:

`(σ , xs) ≅⌊ (σ' , xs') = (xs ⊆ xs' × xs' ⊆ xs) × ∀ x → x ∈ xs → σ x ≡ σ' x`

where list inclusion $\subseteq$ is defined in the standard library by: `xs ⊆ ys = ∀ {x} → x ∈ xs → x ∈ ys`. As a special case of equality on restrictions we define: `σ ≅σ' ⌊xs = (σ, xs) ≅⌊(σ' , xs)`. Also, let composition of substitutions be defined by:

`(σ ⊙ σ') x = σ' x • σ`

Then we have the next results which are extended from [9, 26, 30]:

Lemma 2.6. *(i)* `subEqRes : ∀ {M σ σ'} → σ ≅ σ' ⌊ fv M → M • σ ≡ M • σ'`
(ii) `updFresh : ∀ {x M N σ} → x # M → M • σ , x := N ≡ M • σ`
(iii) `composRenUpd : ∀ {x z M N σ} → z ∉ fv M - x`
`→ M • σ , x := N ≡ M [ x := v z ] • σ , z := N`
(iv) `subDistribUpd : ∀ {M N σ x} → M • σ , x := (N • σ) ≡ M [ x := N ] • σ`
(v) `subComp : ∀ {M σ σ'} → M • σ • σ' ≡ M • σ' ⊙ σ`

2.4 Alpha-conversion

We define α-conversion inductively by:

```
data _∼α_ : Λ → Λ → Set where
  ∼c : ∀ {k} → c k ∼α c k
  ∼v : ∀ {x} → v x ∼α v x
  ∼· : ∀ {M M' N N'} → M ∼α M' → N ∼α N' → M · N ∼α M' · N'
  ∼λ : ∀ {x x' y A A' M M'} → A ∼α A' → y ∉ fv M - x → y ∉ fv M' - x'
     → M [ x := v y ] ≡ M' [ x' := v y ] → λ[ x : A ] M ∼α λ[ x' : A' ] M'
  ∼Π : ∀ {x x' y A A' B B'} → A ∼α A' → y ∉ fv B - x → y ∉ fv B' - x'
     → B [ x := v y ] ≡ B' [ x' := v y ] → Π[ x : A ] B ∼α Π[ x' : A' ] B'
```

In the rule for λ-abstractions and Π-types we require that the bodies are syntactical equal once their respective bound names have been replaced by a common fresh name. In former work the premises were stated in terms of the very same relation being defined. Later in this section and by having a robust infrastructure we will show a posteriori that both definitions for α-conversion are equivalent.

By using this formulation we can show the following result which is central in the theory of α-conversion with simpler methods:

Lemma 2.7. `iotaAlpha : ∀ {M M'} → M • ι ≡ M' • ι → M ∼α M'`

Proof. By structural induction on M and subordinate case analysis on M'. We only show the subcase for the λ-abstractions which did not follow by structural induction before. The respective subcase for Π-types is analogous. There, given $\lambda[x : A]M \bullet \iota \equiv \lambda[x' : A']M' \bullet \iota$, we must show that $\lambda[x : A]M \sim\alpha \lambda[x' : A']M'$. We proceed as follows. First, by definition of substitution we have that our hypothesis is definitional equal to $\lambda[z : A \bullet \iota]M[x := z] \equiv \lambda[x' : A' \bullet \iota]M'[x' := z']$ for some sufficiently fresh names z and z'. Next, by injectivity of the constructors we know that $z \equiv z'$, $A \bullet \iota \equiv A' \bullet \iota$ and $M[x := z] \equiv M'[x' := z']$. Then, by the inductive hypothesis on A we have $A \sim\alpha A'$. And finally, by the rule of λ-abstractions we can derive the desired goal. □

Notice that we did not have to use the induction hypothesis on $M[x := y]$. Otherwise we would have to use some method other than structural induction, e.g., complete induction on the length of M as in [26], since $M[x := y]$ is not a proper component of $\lambda[x : A]M$.

Next we have some properties about α-conversion that are extended quite directly from previous work and which we are going to need later on . Let α-conversion be extended to restrictions by:
`σ ∼α σ' ⌊ xs = ∀ x → x ∈ xs → σ x ∼α σ' x`
Then:

Lemma 2.8. *(i)* `compatSubAlpha : ∀ {M M' σ} → M ∼α M' → M • σ ≡ M' • σ`
(ii) `subAlpha : ∀ {M σ σ'} → σ ∼α σ' ⌊ fv M → M • σ ∼α M • σ'`
(iii) $\sim\alpha$ *is an equivalence.*
(iv) `composRenUnary : ∀ {x y σ M N} → y #⌊ (σ , fv M - x)`
`→ (M • σ , x := v y) [ y := N ] ∼α M • σ , x := N`

Lemma 2.8.(i) not only states that substitution is compatible with α-conversion, but also that it *equalizes* α-convertible λ-terms due to the uniform renaming of the bound variables. Following [25], we will also say σ puts λ-terms into σ-normal form with respect to α-conversion or simply into σ-normal form.

Note that Lemma 2.8.(iv) cannot be strengthened up to syntactical equality. On the left-hand side of $\sim\alpha$ we have that the image of σ for every name $z \not\equiv x$ is being ι-normalized by the substitution $\iota, y := N$ (which has the same effect as ι since y is fresh for every image in σ by definition), while on the right-hand side it is not. This observation will have some repercussions on the proof of closure under substitution of the type system presented in Section 4.

2.4.1 Adequacy of Alpha-conversion

Let $\sim\alpha_s$ be defined as $\sim\alpha$ except that in the clause for λ-abstractions and Π-types we have the following premises for the bodies: $M[x := y] \sim\alpha_s M'[x' := y]$ and $B[x := y] \sim\alpha_s B'[x' := y]$. It follows that both definitions are extensionally equivalent.

As to the soundness of $\sim\alpha$, first we need the following result which can be easily adapted from previous work since it does not depend on any other result about α-conversion nor substitutions (the proof is conducted by complete induction on the length of M as already commented somewhere):

Lemma 2.9. `iotaAlphaSt : ∀ {M M'} → M • ι ≡ M' • ι → M ∼αₛ M'`

Theorem 2.10. `soundAlpha : ∀ {M N} → M ∼α N → M ∼αₛ N`

Proof. Let $M \sim_\alpha N$. By Lemma 2.8.(i) we have $M \bullet \iota \equiv N \bullet \iota$, so by Lemma 2.9 we have $M \sim\alpha_s N$. □

As to the other direction, first we need the following result:

Lemma 2.11. `alphaEq : ∀ {M M' x x' y} → M [ x := v y ] ∼α M' [ x' := v y ]`
`→ M [ x := v y ] ≡ M' [ x' := v y ]`

Proof. Note that by definition of equality on restrictions we have that $(\iota, x := y) \cong \iota \odot (\iota, x := y) \restriction xs$ for any x, y and xs. Then we can reason in the following way:

$$\begin{aligned}
M[x := y] &\equiv M \bullet \iota \odot (\iota, x := y) && \text{by Lemma 2.6.(i)}\\
&\equiv M[x := y] \bullet \iota && \text{by Lemma 2.6.(v)}\\
&\equiv M'[x' := y] \bullet \iota && \text{by Lemma 2.8.(i)}\\
&\equiv M' \bullet \iota \odot (\iota, x' := y) && \text{by Lemma 2.6.(v)}\\
&\equiv M'[x' := y] && \text{by Lemma 2.6.(i)}
\end{aligned}$$

□

Theorem 2.12. `completeAlpha : ∀ {M N} → M ∼α_s N → M ∼α N`

Proof. By structural induction on the derivation of $M \sim\alpha_s N$. The only interesting cases are that of λ-abstractions and Π-types. There, one has to use Lemma 2.11 to obtain the desired premises. □

2.5 Beta-conversion

Given any binary relation on the λ-terms, `_𝒮_`, we define its contextual closure by:

```
data _→C_ : Λ → Λ → Set where
  →cxt : ∀ {M N} → M 𝒮 N → M →C N
  →λR  : ∀ {x M M' A} → M →C M' → λ[ x : A ] M →C λ[ x : A ] M'
  →ΠR  : ∀ {x B B' A} → B →C B' → Π[ x : A ] B →C Π[ x : A ] B'
  →λL  : ∀ {x M A A'} → A →C A' → λ[ x : A ] M →C λ[ x : A' ] M
  →ΠL  : ∀ {x B A A'} → A →C A' → Π[ x : A ] B →C Π[ x : A' ] B
  →·L  : ∀ {M N P} → M →C N → M · P →C N · P
  →·R  : ∀ {M N P} → M →C N → P · M →C P · N
```

β-contraction (`▷β`) is defined as the inductive relation with the single rule: $\lambda[x:A]M \cdot N \triangleright\beta\ M[x := N]$. Then one-step β-reduction (`→β`) is defined as its contextual closure: `_→β_ = _→C_ _▷β_`. Many-step β-reduction (`↠β`) and β-conversion (`≃β`) are respectively defined as the symmetric-and-transitive and equivalence closure of β-reduction augmented with α-conversion: `_→β*_ = Star (_∼α_ ∪ _→β_)` and `_≃β_ = EqClosure (_∼α_ ∪ _→β_)`.

Our definition of one-step β-reduction is compatible with substitution only up to α-conversion as in Hindley and Seldin's treatment, thought there this result is left implicit. The proof is adapted from [30]:

Lemma 2.13. `compatRedSub : ∀ {M N σ} → M →β N → ∃ λ P → M • σ →β P × P ∼α N • σ`

Many-step β-reduction and β-conversion, on the other hand, are "fully" closed under substitution and their proofs follow by structural induction on the derivations of the respective hypotheses, and which are also extended from the aforementioned cite:

Lemma 2.14. *(i)* `compatRedsSub : ∀ {M N σ} → M →β* N → M • σ →β* N • σ`
(ii) `compatConvSub : ∀ {M N σ} → M ≃β N → M • σ ≃β N • σ`

3 The Pure Type Systems

Let $\mathscr{A}$, the *axioms*, be any binary relation on $\mathscr{C}$, and let $\mathscr{R}$, the *rules*, be any 3-ary relation on $\mathscr{C}$ as well. Whenever some s_1 is related to some other s_2 under $\mathscr{A}$ we shall write $\mathscr{A}\, s_1\, s_2$ as is the case for infix relations in most implementations of type theory, e.g., Agda. We will do analogously for $\mathscr{R}$.

The PTS is inductively defined in Figure 2, where $\Gamma, x : A$ is definitionally equal to $(x, A) :: \Gamma$. We roughly follow the presentation in Section 4.4.10 of [23]. We have two kinds of judgments mutually defined and with the following meaning: (i) Γ ok means that Γ is a valid or well-formed context, and; (ii) $\Gamma \vdash M : A$ that M has type A under the context Γ.

To be more precise, the type system is defined by using *generalized* induction [20, p. 382]. The rules ⊢abs and ⊢prod have infinitely many branching trees due to the scope of the quantification in the premises concerning the body of the abstractions therein. Having a derivation for every fresh name will give us a stronger induction hypothesis in the cases at issue for the proof of the thinning lemma, which otherwise we would have to prove by complete induction on the length of the derivations or by using some equivariance result (renaming lemmas). A more up-to-date version of this technique is known as

$$\vdash\text{nil}\ \frac{}{[]\ \mathsf{ok}} \qquad \vdash\text{cons}\ \frac{\Gamma\,\mathsf{ok} \qquad \Gamma \vdash A : s}{\Gamma, x : A\,\mathsf{ok}}\ (x \notin \mathsf{dom}\,\Gamma)$$

$$\vdash\text{sort}\ \frac{\Gamma\,\mathsf{ok}}{\Gamma \vdash s_1 : s_2}\ (\mathscr{A}\, s_1\, s_2)$$

$$\vdash\text{prod}\ \frac{\Gamma \vdash A : s_1 \qquad \forall y \to y \notin \mathsf{dom}\,\Gamma \to \Gamma, y : A \vdash B[x := y] : s_2}{\Gamma \vdash \Pi[x : A]B : s_3}\ (\mathscr{R}\, s_1\, s_2\, s_3)$$

$$\vdash\text{var}\ \frac{\Gamma\,\mathsf{ok}}{\Gamma \vdash x : A}\ ((x, A) \in \Gamma)$$

$$\vdash\text{abs}\ \frac{\Gamma \vdash A : s_1 \quad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z : A \vdash B[y := z] : s_2 \qquad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z : A \vdash M[x := z] : B[y := z]}{\Gamma \vdash \lambda[x : A]M : \Pi[y : A]B}\ (\mathscr{R}\, s_1\, s_2\, s_3)$$

$$\vdash\text{app}\ \frac{\Gamma \vdash M : \Pi[x : A]B \qquad \Gamma \vdash N : A \qquad \Gamma \vdash B[x := N] : s}{\Gamma \vdash M \cdot N : B[x := N]}$$

$$\vdash\text{conv}\ \frac{\Gamma \vdash M : A \qquad A \simeq_\beta B \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$$

Figure 2: Pure Type Systems

cofinite quantification. We note that Agda does not give any special meaning to generalized induction and it is treated just as any other inductive clause. We will also refer to these kind of rules as *infinitary*.

The occurrence of the premise $\Gamma \vdash A : s_1$ in the rule $\vdash$prod is standard, e.g., see [3]. On the other hand, having $\Gamma \vdash B[x := N] : s$ in the $\vdash$app rule is new to the best of our knowledge, and as we shall see in due course, it is convenient for the proof of the substitution lemma when using Stoughton's definition.

3.1 Basic Properties

To begin with, we have that variables mentioned anywhere must be declared:

Lemma 3.1. *(i)* `fvCxt : ∀ {Γ x A w} → Γ ok → (x , A) ∈ Γ → w * A → w ∈ dom Γ`
(ii) `fvAsg : ∀ {Γ M A w} → Γ ⊢ M : A → x * M · A → w ∈ dom Γ`

Proof. By simultaneous induction on the the structure of the typing derivation and subordinate case analysis on the occurrence of w. The only interesting case is that of λ-abstractions; the others follow either analogously or directly by the IH. There, we have the hypotheses:

$$\text{(a)}\ \frac{\Gamma \vdash A : s_1 \quad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z : A \vdash B[y := z] : s_2 \qquad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z : A \vdash M[x := z] : B[y := z]}{\Gamma \vdash \lambda[x : A]M : \Pi[y : A]B}\ (\mathscr{R}\, s_1\, s_2\, s_3)$$

and (b) $w \in \mathsf{fv}\,(\lambda[x : A]M +\!\!+ \Pi[x : A]B)$, and we must show $w \in \mathsf{dom}\,\Gamma$. By Lemma 2.1 there are three different cases as to the generation of (b):

- Case $w * A$. Immediate by the IH (ii).
- Case $w * M$ and $x \not\equiv w$. Let $z = X'(w :: \mathsf{dom}\,\Gamma)$. By Lemma 2.3 we have that $z \notin w :: \mathsf{dom}\,\Gamma$, thus $z \notin \mathsf{dom}\,\Gamma$ and $z \not\equiv w$. Next, since $w * w[x := z]$ then we have $w * M[x := z]$ by Lemma 2.5. Finally, by the IH (ii) using the bottom-most premise we have $w \in \mathsf{dom}\,z :: \Gamma$, hence $w \in \mathsf{dom}\,\Gamma$.

- Case $w * B$ and $y \not\equiv w$. Analogous to the previous case. □

The next result follows directly as the contrapositive of the previous lemma:

Corollary 3.1.1. *(i)* `freshCxt : ∀ {Γ y A w} → Γ ok → w ∉ dom Γ → (y , A) ∈ Γ → w # A`
(ii) `freshAsg : ∀ {Γ M A w} → w ∉ dom Γ → Γ ⊢ M : A → w # M · A`

Lastly, the following results about the validity of contexts and generation of Π-types are routine:

Lemma 3.2. *(i)* `validCxt : ∀ {Γ M A} → Γ ⊢ M : A → Γ ok`
(ii) `genProd : ∀ {Γ x A B C} → Γ ⊢ Π[ x : A ] B : C → ∃₃ λ s₁ s₂ s₃ → 𝓡 s₁ s₂ s₃ × Γ ⊢ A : c s₁ × (∀ y → y ∉ dom Γ → Γ , y : A ⊢ B [ x := v y ] : c s₂) × C ≃β c s₃`

4 Some Fundamental Metatheory

The type system presented in the previous section enjoys some nice metatheoretic properties such as thinning (Lemma 4.1), syntactic validity (Lemma 4.3), closure under α-conversion (Lemma 4.4) and substitution (Lemma 4.6).

The thinning lemma follows quite easily thanks to the use of generalized induction.

As to the other results, the situation is a bit more complicated. Consider a presentation of the type system in which the application rule does not mention the third premise, and let use proceed to prove informally, to begin with, syntactic validity and by using structural induction. In the complex case of applications we are given:

$$\frac{\Gamma \vdash M : \Pi[x:A]B \qquad \Gamma \vdash N : A}{\Gamma \vdash M \cdot N : B[x := N]} \tag{1}$$

and we have to show that $B[x := N]$ is a valid type, i.e., either $\Gamma \vdash B[x := N] : s$ or $B[x := N] \equiv s$ for some sort s. First, by the induction hypothesis we can derive that $\Gamma \vdash \Pi[x : A]B : s$. Then, by the generation lemma we get $\Gamma, y : A \vdash B[x := y] : s$ for every fresh name y. Now, to derive the goal from this last result we have to apply the substitution $\iota, y := N$ and obtain a valid type, all of which requires having some substitution lemma at hand. So far is routine.

Next let us consider the proof of closure under substitution of the type system. Again, let us take a look at the case of applications. Given (1), we have to show $\Delta \vdash (M \cdot N) \bullet \sigma : B[x := N] \bullet \sigma$ for some context Δ. By the induction hypothesis on each premise followed by the application rule one can derive $\Delta \vdash (M \cdot N) \bullet \sigma : (B \bullet \sigma, x := x')[x' := N \bullet \sigma]$. Now, the types $(B \bullet \sigma, x := x')[x' := N \bullet \sigma]$ and $B[x := N] \bullet \sigma$ are not identical but α-convertible, as already commented somewhere, so it is necessary to have closure under α-conversion at hand. In [19, 20, 23] the authors were able to prove directly the lemma because there substitution preserves the identity of the λ-terms. However, as we shall discuss later on, they had to resort to more involved methods than us for an important lemma in the theory of conversion.

An alternative to using closure under α-conversion above is to use the conversion rule and rewrite the types at question, however, this is not easy. In that case, one should first build a derivation of the (syntactic) validity of $(B \bullet \sigma, x := x')[x' := N \bullet \sigma]$. The direct procedure to this end is to use the validity lemma on the left-hand side premise in (1), then the generation lemma to obtain $B[x := y]$ for some fresh name y and finally the induction hypothesis twice, once with $\sigma, y := z$ and other time with $\iota, z := N \bullet \sigma$ for some sufficiently fresh name z (which might perfectly be the same name as y), all of which yields a result identical to the one desired because of Lemma 2.6.(iii).[1]. Now, the main problem with this argument is

[1] Note that we do not have a result for composing well-typed substitutions, a result which seems to require the very same lemma we are trying to prove.

that it is not always the case that the derivation of the validity of $B[x := y]$ constructed during the proof of the validity lemma is of a smaller size in any way than that of (1), hence the well-foundedness of the whole argument is questioned. A more elaborated method is required.

So, back to our point, the proof of closure under α-conversion also relies on the previous lemmas. In the case of applications we have $M \cdot N \sim\alpha M' \cdot N'$, in addition to (1), and we must find a derivation of $\Delta \vdash M' \cdot N' : B[x := N]$ for some context Δ. By the induction hypotheses followed by the application rule we can derive $\Delta \vdash M' \cdot N' : B[x := N']$. Now, to rewrite the type $B[x := N']$ into $B[x := N]$, first we have to show that the latter is a valid type. This only seems possible if we use syntactic validity and then substitution, and so closing the dependency circle.

Now, instead of attempting to prove all three lemmas simultaneously, we will make a slight simplification to the problem. By adding the premise $\Gamma \vdash B[x := N] : s$ to the application rule we have managed to cut two of the dependencies: α-conversion on syntactic validity, and the latter on substitution. As a result, we shall see that syntactic validity and α-conversion can be proven separately and first, and substitution after. Furthermore, we shall prove that said premise is derivable in the sense that we can consider a presentation of the type system that does not mention it yet derives the same judgments.

4.1 Thinning

To begin with, we have the thinning lemma which follows in a fairly direct way (we refer the reader to [23, p. 65] for an account on the proof):

Lemma 4.1. `thinning : ∀ {Γ Δ M A} → Γ ⊆ Δ → Δ ok → Γ ⊢ M : A → Δ ⊢ M : A`

4.2 Syntactic Validity

Every type assignable to some λ-term is itself a valid type. For many presentations, e.g., [3, 13, 19, 20, 23], this result has to wait until closure under substitution has been established. In our case, however, because we have added the aforementioned premise we can show it in advance.

The proof is really straightforward and we do not need to use induction, just case analysis. We only need the next lemma for the case of variables and which follows by structural induction on the derivation of Γok and by using the thinning lemma:

Lemma 4.2. `validDecl : ∀ {Γ x A} → Γ ok → (x , A) ∈ Γ → ∃ λ s → Γ ⊢ A : c s`

Lemma 4.3. `syntacticValidity : ∀ {Γ M A} → Γ ⊢ M : A → ∃ λ s → A ≡ c s ⊎ Γ ⊢ A : c s`

4.3 Closure Under Alpha-conversion

Next we have closure under α-conversion. We are going to split the lemma in half, one for the conversion of subjects and the other for the predicates.

Let α-conversion be extended to contexts pointwise by:

`_≈α_ = Pointwise (λ (x , A) (y , B) → x ≡ y × A ∼α B)`

Then we have the first part:

Lemma 4.4. *(i)* `closAlphaCxt : ∀ {Γ Δ} → Γ ≈α Δ → Γ ok → Δ ok`
(ii) `closAlphaAsg : ∀ {Γ Δ M N A} → Γ ≈α Δ → M ∼α N → Γ ⊢ M : A → Δ ⊢ N : A`

Proof. By simultaneous induction on the structure of the typing derivation. We only show some cases:

- Case ⊢abs. We have:

$$\frac{\Gamma \vdash A : s_1 \quad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z : A \vdash B[y := z] : s_2 \quad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z : A \vdash M[x := z] : B[y := z]}{\Gamma \vdash \lambda[x : A]M : \Pi[y : A]B}\,(\mathscr{R}\, s_1\, s_2\, s_3)$$

and $\lambda[x : A]M \sim\alpha\ \lambda[x' : A']M'$ which follows from $A \sim\alpha\ A'$ and $M[x := w] \equiv M'[x' := w]$ for some w not in $\mathsf{fv}M - x$ nor in $\mathsf{fv}M' - x'$. We have to show: $\Delta \vdash \lambda[x' : A']M' : \Pi[y : A]B$. To use the λ-abstraction rule to derive $\Delta \vdash \lambda[x' : A']M' : \Pi[y : A']B$, and so then use the conversion rule to rewrite $\Pi[y : A']B$ into $\Pi[y : A]B$, first we need to show:

(a) $\Delta \vdash A : s_1$;

(b) $\Delta, z : A' \vdash B[y := z] : c_2$ for all $z \notin \mathsf{dom}\,\Delta$, and;

(c) $\Delta, z : A' \vdash M[x := z] : B[y := z]$ for all $z \notin \mathsf{dom}\,\Delta$.

(a) is immediate by the IH (ii). As to (b) and (c), we show only the latter since the other is analogous. Let z be some name not in $\mathsf{dom}\,\Delta$. Then we also have that $z \notin \mathsf{dom}\,\Gamma$ by definition of $\approx\alpha$. Next, by congruence on one of the hypotheses we have $M[x := w][w := z] \equiv M'[x' := w][w := z]$, and by Lemma 2.6.(iii) we can obtain $M[x := z] \equiv M'[x' := z]$. Then, since α-conversion is reflexive, we have $M[x := z] \sim\alpha M'[x' := z]$, so we can use the IH (ii) and get $\Delta \vdash \lambda[x' : A']M' : \Pi[y : A']B$. Now, it is easy to show that $\Pi[y : A]B \sim\alpha\ \Pi[y : A']B$, hence by the IH (ii) we have $\Delta \vdash \Pi[y : A']B$, and so we can use the conversion rule as explained earlier.

- Case $\vdash$app. There we have:

$$\frac{\Gamma \vdash M : \Pi[x : A]B \qquad \Gamma \vdash N : A \qquad \Gamma \vdash B[x := N] : s}{\Gamma \vdash M \cdot N : B[x := N]}$$

$M \sim\alpha\ M'$ and $N \sim\alpha\ N'$, and we must show $\Delta \vdash M' \cdot N' : B[x := N]$. First, note that $\iota, x := N$ and $\iota, x := N'$ assigns α-convertibles terms to every variable, thus we have $\iota, x := N \sim\alpha\ \iota, x := N' \restriction \mathsf{fv}B$. Then, by Lemma 2.8.(ii) we obtain $B[x := N'] \sim\alpha\ B[x := N]$. Now, by the IH (ii) on each premise followed by the application rule we obtain $\Delta \vdash M' \cdot N' : B[x := N']$. Finally, by the IH (ii) again we can derive $\Delta \vdash B[x := N] : s$ and so we can rewrite the types to obtain the desired goal. □

As to the second part of the lemma we have that predicates are closed under α-conversion as well. Its proof follows directly by using closure of the subjects and syntactic validity:

Corollary 4.4.1. `closAlphaPr : ∀ {Γ Δ M A B} → Γ ∼αs Δ → A ∼α B → Γ ⊢ M : A → Δ ⊢ M : B`

4.4 Closure Under Substitution

Having established thinning, syntactic validity and closure under α-conversion, we are almost ready to prove the substitution lemma. But first, some quick preparatory definitions. Let us define well-typed substitutions from variables in Γ to terms of appropriate type under Δ by:

`σ : Γ ⇀ Δ = ∀ {x A} → (x , A) ∈ Γ → Δ ⊢ σ x : A • σ`

The next result will provide us the required hypothesis to invoke the induction hypothesis in the case of λ-abstractions and Π-types:

Lemma 4.5. `subRen : ∀ {Γ Δ x y s A σ} → x ∉ dom Γ → y ∉ dom Δ → Γ ⊢ A : c s → Δ ⊢ A • σ : c s → σ : Γ ⇀ Δ → (σ , x := v y) : (Γ , x : A) ⇀ (Δ , y : A • σ)`

Proof. By definition of well-typed substitutions, we must show that for any declaration $(z,B) \in \Gamma, x:A$, it follows that $\Delta, y:A \bullet \sigma \vdash (\sigma, x := y)z : B \bullet (\sigma, x := y)$. First, notice that by Lemma 3.2.(i) we have $\Delta\,\mathsf{ok}$. Also, since $y \notin \mathsf{dom}\,\Delta$, then by the $\vdash$cons rule we have $\Delta, y:A \bullet \sigma\,\mathsf{ok}$ as well. Next we analyze whether z is equal to x or not:

- Case $z \equiv x$. Then $A \equiv B$ must also be the case, so the goal becomes $\Delta, y:A \bullet \sigma \vdash y : A \bullet \sigma, x := y$. By $\vdash$var we have $\Delta, y:A \bullet \sigma \vdash y : A \bullet \sigma$. Then, by the Corollary 3.1.1 we have $x\#A$, hence by Lemma 2.6.(ii) we know that $A \bullet \sigma, x := y \equiv A \bullet \sigma$, and so we can rewrite the type in the previous derivation into $A \bullet \sigma, x := y$ to obtain our goal.
- Case $z \not\equiv x$. Then $(z,B) \in \Gamma$ and we must show $\Delta, y:A \bullet \sigma \vdash \sigma z : B \bullet \sigma, x := y$. First, by hypothesis we have $\Delta \vdash \sigma z : A \bullet \sigma$, so by thinning we also have $\Delta, y:A \bullet \sigma \vdash \sigma z : B \bullet \sigma, x := y$. Then we can proceed analogously to the previous case and rewrite the type $B \bullet \sigma, x := y$ into $B \bullet \sigma$. □

Then we have that typing is closed under well-typed substitutions:

Lemma 4.6. `closureSub : ∀ {Γ Δ M A σ} → σ : Γ ⇀ Δ → Δ ok → Γ ⊢ M : A → Δ ⊢ M • σ : A • σ`

Proof. By simultaneous induction on the typing derivation. We only show some cases:

- Case $\vdash$abs. We have:

$$\frac{\Gamma \vdash A : s_1 \quad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z:A \vdash B[y := z] : s_2 \qquad \forall z \to z \notin \mathsf{dom}\,\Gamma \to \Gamma, z:A \vdash M[x := z] : B[y := z]}{\Gamma \vdash \lambda[x:A]M : \Pi[y:A]B}\,(\mathscr{R}\, s_1\, s_2\, s_3)$$

  and we must derive $\Delta \vdash \lambda[x' : A \bullet \sigma](M \bullet \sigma, x := x') : \Pi[y' : A \bullet \sigma](B \bullet \sigma, y := y')$, where x' and y' are definitional equal to $X(\sigma, \mathsf{fv}\,M - x)$ and $X(\sigma, \mathsf{fv}\,B - y)$ respectively. To use the λ-abstraction rule to derive our goal first we need to show:

  (a) $\Delta \vdash A \bullet \sigma : s_1$

  (b) $\Delta, z{:}A \bullet \sigma \vdash (B \bullet \sigma, y := y')[y' := z] : c_2$ for all $z \notin \mathsf{dom}\,\Delta$, and;

  (c) $\Delta, z{:}A \bullet \sigma \vdash (M \bullet \sigma, x{:=}x')[x' := z] : (B \bullet \sigma, y := y')[y' := z]$ for all $z \notin \mathsf{dom}\,\Delta$.

  (a) is immediate from the IH. As to (b) and (c), again, we only show the latter. Let z be any name not in Δ and let $w = X'(\mathsf{dom}\,\Gamma)$. By Lemma 2.3 we have $w \notin \mathsf{dom}\,\Gamma$, thus by Lemma 4.5 $(\sigma, w := z) : (\Gamma, w : A) \rightharpoonup (\Delta, z : A \bullet \sigma)$. Next, by the IH with the previous result we have:

$$\Delta, z : A \bullet \sigma \vdash M[x := w] \bullet (\sigma, w := z) : B[y := w] \bullet (\sigma, w := z) \tag{2}$$

  Now, to derive our goal it suffices to show that the subjects in the goal (b) and in (2) are α-convertible, and similarly with the predicates therein. As to the subjects, first, by Lemma 3.1.1 we note that $w \notin \mathsf{fv}\,M - x$, thus by Lemma 2.6.(ii) we have $M[x := w] \bullet (\sigma, w := z') \equiv M \bullet (\sigma, x := z')$. And second, by Lemma 2.4 we have $x'\#\downarrow(\sigma, \mathsf{fv}\,M - x)$, hence we can use Lemma 2.8.(iv) and derive that $M \bullet (\sigma, x := z) \sim\alpha\ (M \bullet \sigma, x := x')[x' := z]$. We can reason analogously to show that types are also α-convertible. Finally, by closure under α-conversion we can obtain (b).

- Case $\vdash$app. We have:

$$\frac{\Gamma \vdash M : \Pi[x:A]B \qquad \Gamma \vdash N : A \qquad \Gamma \vdash B[x := N] : s}{\Gamma \vdash M \cdot N : B[x := N]}$$

$$\vdash\text{nil}\ \frac{}{[]\,\mathsf{ok}} \qquad \vdash\text{cons}\ \frac{\Gamma\,\mathsf{ok} \qquad \Gamma \vdash A : s}{\Gamma, x : A\,\mathsf{ok}}\ (x \notin \mathsf{dom}\,\Gamma)$$

$$\vdash\text{sort}\ \frac{\Gamma\,\mathsf{ok}}{\Gamma \vdash s_1 : s_2}\ (\mathscr{A}\, s_1\, s_2)$$

$$\vdash\text{prod}\ \frac{\Gamma \vdash_{\mathsf{s}} A : s_1 \qquad \Gamma, y : A \vdash_{\mathsf{s}} B[x := y] : s_2}{\Gamma \vdash_{\mathsf{s}} \Pi[x : A]B : s_3}\ \begin{cases} \mathscr{R}\, s_1\, s_2\, s_3 \\ y \notin \mathsf{fv}\, B - x \end{cases}$$

$$\vdash\text{var}\ \frac{\Gamma\,\mathsf{ok}}{\Gamma \vdash x : A}\ ((x, A) \in \Gamma)$$

$$\vdash\text{abs}\ \frac{\Gamma \vdash_{\mathsf{s}} A : s_1 \qquad \Gamma, z : A \vdash_{\mathsf{s}} B[y := z] : s_2 \qquad \Gamma, z : A \vdash_{\mathsf{s}} M[x := z] : B[y := z]}{\Gamma \vdash_{\mathsf{s}} \lambda[x : A]M : \Pi[y : A]B}\ \begin{cases} \mathscr{R}\, s_1\, s_2\, s_3 \\ z \notin \mathsf{fv}\, M - x \\ z \notin \mathsf{fv}\, B - y \end{cases}$$

$$\vdash\text{app}\ \frac{\Gamma \vdash_{\mathsf{s}} M : \Pi[x : A]B \qquad \Gamma \vdash_{\mathsf{s}} N : A}{\Gamma \vdash_{\mathsf{s}} M \cdot N : B[x := N]}$$

$$\vdash\text{conv}\ \frac{\Gamma \vdash M : A \qquad A \simeq_\beta B \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}$$

Figure 3: Standard (Finitary) PTS

and we must show $\Delta \vdash M \cdot N \bullet \sigma : B[x := N] \bullet \sigma$. By the IH on each premise followed by the application rule we have $\Delta \vdash (M \cdot N) \bullet \sigma : (B \bullet \sigma, x := x')[x' := N \bullet \sigma]$ where $x' = X(\sigma, \mathsf{fv}\, B - x)$. Next, we can use Lemma 2.8.(iv) and derive $(B \bullet \sigma, x := x')[x' := N \bullet \sigma] \sim_\alpha B \bullet \sigma, x := (N \bullet \sigma)$. Finally, by Lemma 2.6.(iv) we have $B \bullet \sigma, x := (N \bullet \sigma) \equiv B[x := N] \bullet \sigma$ so by closure under α-conversion we obtain our goal.

- Case $\vdash$conv. By the IH and Lemma 2.14.(ii). □

As a particular case of the substitution lemma we have the following cut result, which will turn out to be helpful in the following section. First, let `_••_ : Cxt → Sub → Cxt` be the operation that extends substitution to contexts pointwise. We have the next result for unary substitutions (whose proof is analogous to the proof of Lemma 4.5):

Lemma 4.7. `subUnary : ∀ {x Γ N A} → x ∉ dom Γ → Γ ⊢ A → Γ •• ι ⊢ N : A • ι → (ι , x := N) : (Γ , x : A) ⇀ Γ •• ι`

Then we have the cut lemma:

Lemma 4.8. `cut : ∀ {Γ M N A B x} → Γ , x : A ⊢ M : B → Γ ⊢ N : A → Γ ⊢ M [ x := N ] : B [ x := N ]`

Proof. By Lemma 4.3, closure under α-conversion, Lemma 4.7, context validity and Lemma 4.6. □

4.5 Adequacy of the Type System

The type system defined in Fig. 2 is equivalent to one which uses finitary rules, i.e., without using generalized induction or cofinite quantification in the $\vdash$prod and $\vdash$abs rules, and furthermore, which does not mention the third premise in the application rule. Fig. 3 shows such a presentation. For a brief discussion on why the freshness conditions in the finitary version are stated relative to the λ-terms therein while in the infinitary one are relative to the context we refer the reader to [23, p. 61].

First, we have that the infinitary version of the system is *sound*:

Theorem 4.9. *(i)* `ptsSound : ∀ {Γ} → Γ ok → Γ ok`$_s$
(ii) `ptsSound : ∀ {Γ M A} → Γ ⊢ M : A → Γ ⊢`$_s$ `M : A`

Proof. By straightforward induction on the structure of the judgments. In the case of products and abstractions, to derive the corresponding premises one has to pick some sufficiently fresh name, e.g., $X'(\mathsf{dom}\,\Gamma)$, and use Corollary 3.1.1 to derive the freshness side-conditions. □

The type system is also *complete*. To prove it, we need the following renaming lemma (whose proof follows similarly to that of Lemma 4.5):

Lemma 4.10. `unaryRen : ∀ {Γ x y A M B} → y ∉ dom Γ → Γ , x : A ⊢ M : B`
`→ Γ , y : A ⊢ M [ x := v y ] : B [ x := v y ]`

Theorem 4.11. *(i)* `ptsComplete : ∀ {Γ} → Γ ok`$_s$ `→ Γ ok`
(ii) `ptsComplete : ∀ {Γ M A} → Γ ⊢`$_s$ `M : A → Γ ⊢ M : A`

Proof. By induction on the structure of the judgments.

- Case ⊢abs. We have:

$$\frac{\Gamma \vdash_{\mathsf{s}} A : s_1 \qquad \Gamma, z : A \vdash_{\mathsf{s}} B[y := z] : s_2 \qquad \Gamma, z : A \vdash_{\mathsf{s}} M[x := z] : B[y := z]}{\Gamma \vdash_{\mathsf{s}} \lambda\,[x : A]M : \Pi[y : A]B}$$

  with $z \notin \mathsf{fv}\,M - x$ and $z \notin \mathsf{fv}\,B - y$, and we must derive $\Gamma \vdash \lambda\,[x : A]M : \Pi[y : A]B$. To use the abstraction rule we have to translate each premise. The left-most one is immediate from the IH (ii). The other two are analogous so we will show only the right-most one. First, by the IH (ii) we have $\Gamma, z : A \vdash M[x := z] : B[y := z]$. Let z' be some name not in $\mathsf{dom}\,\Gamma$. By Lemma 4.10 we have $\Gamma, z' : A \vdash M[x := z][z := z'] : B[y := z][z := z']$, so by Lemma 2.6.(iii) we have the desired result.

- Case ⊢app. We have:

$$\frac{\Gamma \vdash_{\mathsf{s}} M : \Pi[x : A]B \qquad \Gamma \vdash_{\mathsf{s}} N : A}{\Gamma \vdash_{\mathsf{s}} M \cdot N : B[x := N]}$$

  and we must derive $\Gamma \vdash M \cdot N : B[x := N]$. By the IH (ii) we have $\Gamma \vdash M : \Pi[x : A]B$ and $\Gamma \vdash N : A$. Let $z = X'(\mathsf{dom}\,\Gamma)$. By syntactic validity we obtain $\Gamma \vdash \Pi[x : A]B : s$ for some s, so by the generation lemma we derive $\Gamma, z : A \vdash B[x := z] : s$. Now, by cut we have $\Gamma \vdash B[x := z][z := N] : s$, and so by Lemma 2.6.(iii) we can derive the missing premise, $\Gamma \vdash B[x := N] : s$.

□

5 Related Work

McKinna and Pollack [19, 20, 23] put forward in the LEGO proof assistant [18] the mechanization of a great body of knowledge about the metatheory of a generalization of the PTS, the Cumulative Type System (CTS): CR and standardization for β-conversion, subject reduction, decidability of the type system (assuming normlization), etc. The syntax uses two sort of names, one for the free variables and other for the bound ones. Consequently, two definitions of the substitution operation must be given. Since the set of variables are disjoint, it is impossible for name capture to happen. Besides, since substitution does not perform any renaming, the identity of the λ-terms is preserved during the operation. As a result,

α-conversion is seldom used in the whole development. However, since the syntax allows to build λ-term that do not have an ordinary interpretation, i.e., those who mention names from the set of the bound variables which are actually not bound to any λ-abstraction or Π-type, a wellformedness predicate must accompany many results to rule them out from consideration and which becomes rather ubiquitous.

Barras and Werner [6, 7] mechanized a substantial part of the metatheory of the Calculus of Constructions (CC) in Coq culminating with the decidability of the type system. The syntax uses de Bruijn notation. In an unpublished work [5], Barras extended the metatheoretic results to the PTS and CTS. The sources can be found in [27].

In [29], Urban et al. used the Isabelle/HOL system [21] together with the Nominal Datatype Package (NDP) or Nominal Isabelle [28] to mechanize the metatheory of LF [14]. The NDP provides a framework to work with inductive types with binders and their associated induction and recursion principles modulo α-conversion. A limitation of their approach is that the NDP does not yet allow generating executable code, so an implementation for a type-checker cannot be extracted directly. Also, since HOL is founded on classical logic, the results about the decidability of LF were not entirely formalized. The sources for LF and NDP can be found respectively in [4] and in Isabelle's distribution [8].

Next we point out the reader some work at the forefront in the mechanization of type theory; these developments focus on larger object theories, e.g., featuring universes, large elimination, Σ-types, and so on, thus a comparison with our work does not seem relevant yet (all of them use de Bruijn indices): In [1], Abel et al. present a proof of the decidability of conversion for a fragment of Martin-Löf's Type Theory in Agda; Adjedj et al. [2] use Coq to mechanize a proof of the decidability for a type system of similar characteristics to the one above, and; in [24], Sozeau et al. present a partial formalization about the decidability of a considerable part of Coq's kernel (normalization is assumed), written in Coq.

6 Conclusions

We have formalized on machine some interesting body of knowledge for the PTS using conventional syntax, i.e., first-order with one-sorted variables names and without identifying α-convertible λ-terms. Among other results we have proven: weakening, syntactic validity, closure under α-conversion and substitution. In the course we had to update the existing framework of Stoughton's substitutions with Church-style λ-abstractions and Π-types. We have also given a new definition for α-conversion that works better than previous ones in that it allowed us to prove some key lemmas by using simpler methods. Except for the adequacy of this new definition, which is not used in the whole development, all results follow by structural induction on the various relations defined.

The use of conventional syntax allowed us to prove closure under substitution of conversion directly by using structural induction. In contrast, McKinna and Pollack had to give an alternative definition using infinitary rules. The proof that both characterizations define the same relation follows by a multiple renaming lemma. This is also the case for the adequacy of the type system. In our case, we did not prove adequacy for conversion since our characterization is almost the same as the one appearing in common textbooks. As to the adequacy of the type system, we have been able to reuse the substitution lemma.

The resulting Agda code is within the limits of what is manageable. The entire development spans about 3000LOC and it is divided equally between the framework and the metatheory of the PTS. To put in perspective, the work by Urban et al. is about 15KLOC, from which 1800LOC belongs to the metatheory up to syntactic validity, a bit larger than our counterpart.The NDP on the other hand is over 9300LOC.As to the work by Barras on the PTS in [5] (which does not address the problem of names at all since it uses de Bruijn indices), the entire corresponding formalization is approximately 2600LOC.

References

[1] Andreas Abel, Joakim Öhman & Andrea Vezzosi (2018): *Decidability of conversion for type theory in type theory*. In: *Proc. ACM Program. Lang.*, 2, pp. 23:1–23:29, doi:10.1145/3158111.

[2] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot & Loïc Pujet (2024): *Martin-Löf à la Coq*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2024, Association for Computing Machinery, New York, NY, USA, p. 230–245, doi:10.1145/3636501.3636951.

[3] H. P. Barendregt (1992): *Lambda calculi with types*. In Abramsky, Gabbai & Maibaum, editors: *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*, Oxford University Press, Inc., USA, p. 117–309, doi:10.1093/oso/9780198537618.003.0002.

[4] Bruno Barras: *Sources of the PTS formalization*. Available at `https://github.com/rocq-archive/pts/tree/v8.5`.

[5] Bruno Barras: *Type-checking PTS*. Available at `https://www.lix.polytechnique.fr/~barras/pts_proofs/PTS/main.html`. Webpage.

[6] Bruno Barras (1996): *Coq en Coq*. Rapport de Recherche 3026, INRIA.

[7] Bruno Barras & Benjamin Werner: *Coq in Coq*. Available at `https://www.lix.polytechnique.fr/~barras/download/coqincoq.ps.gz`. Unpublished manuscript.

[8] University of Cambridge & Technische Universität München: *Isabelle 2016 Homepage*. Available at `https://isabelle.in.tum.de/website-Isabelle2016-1/index.html`.

[9] Ernesto Copello, Nora Szasz & Álvaro Tasistro (2016): *Formal metatheory of the Lambda calculus using Stoughton's substitution*. *Theoretical Computer Science* 685, doi:10.1016/j.tcs.2016.08.025.

[10] Martín Copes, Nora Szasz & Álvaro Tasistro (2018): *Formalization in Constructive Type Theory of the Standardization Theorem for the Lambda Calculus using Multiple Substitution*. In Frédéric Blanqui & Giselle Reis, editors: *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*, *EPTCS* 274, pp. 27–41, doi:10.4204/EPTCS.274.3.

[11] H.B. Curry & R. Feys (1958): *Combinatory Logic*. *Combinatory Logic* v. 1, North-Holland Publishing Company. Available at `https://books.google.com.uy/books?id=fEnuAAAAMAAJ`.

[12] Jean-Yves Girard, Paul Taylor & Yves Lafont (1989): *Proofs and types*. Cambridge University Press, USA.

[13] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A framework for defining logics*. *J. ACM* 40(1), p. 143–184, doi:10.1145/138027.138060.

[14] Robert Harper & Frank Pfenning (2005): *On equivalence and canonical forms in the LF type theory*. *ACM Trans. Comput. Logic* 6(1), p. 61–101, doi:10.1145/1042038.1042041.

[15] J. Roger Hindley & Jonathan P. Seldin (1986): *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press.

[16] F. Joachimski & R. Matthes (2003): *Short Proofs of Normalization for the simply-typed lambda-calculus, permutative conversions and Gödel's T*. *Archive for Mathematical Logic* 42, p. 59–87, doi:10.1007/s00153-002-0156-9.

[17] Ryo Kashima (2000): *A Proof of the Standardization Theorem in Lambda-Calculus*. Technical Report C-145, Research Reports on Mathematical and Computing Sciences, Tokyo Institute of Technology.

[18] Zhaohui Luo & Robert Pollack (1992): *LEGO Proof Development System: User's Manual*. Technical Report, University of Edinburgh. Available at `http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-211`.

[19] James McKinna & Robert Pollack (1993): *Pure type systems formalized*. In Marc Bezem & Jan Friso Groote, editors: *Typed Lambda Calculi and Applications*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 289–305, doi:10.1007/BFb0037113.

[20] James McKinna & Robert Pollack (1999): *Some Lambda Calculus and Type Theory Formalized*. Journal of Automated Reasoning 23(3), pp. 373–409, doi:10.1023/A:1006294005493.

[21] Tobias Nipkow, Markus Wenzel & Lawrence C. Paulson (2002): *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, doi:10.1007/3-540-45949-9.

[22] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Chalmers University of Technology. Available at `https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf`.

[23] Robert Pollack (1994): *The Theory of LEGO*. Ph.D. thesis, University of Edinburgh. Available at `https://era.ed.ac.uk/handle/1842/504`.

[24] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau & Théo Winterhalter (2025): *Correct and Complete Type Checking and Certified Erasure for Coq, in Coq*. J. ACM 72(1), doi:10.1145/3706056.

[25] Allen Stoughton (1988): *Substitution revisited*. Theoretical Computer Science 59(3), pp. 317–325, doi:10.1016/0304-3975(88)90149-1.

[26] Álvaro Tasistro, Ernesto Copello & Nora Szasz (2015): *Formalisation in Constructive Type Theory of Stoughton's Substitution for the Lambda Calculus*. In Mauricio Ayala-Rincón & Ian Mackie, editors: *Proc. LSFA '14*, *ENTCS* 312, Elsevier, pp. 215–230, doi:10.1016/j.entcs.2015.04.013.

[27] Christian Urban: *Sources of the paper: Mechanizing the Metatheory of LF*. Available at `http://nms.kcl.ac.uk/christian.urban/Nominal/LF/LF.tgz`.

[28] Christian Urban & Stefan Berghofer: *Nominal Isabelle*. Available at `https://isabelle.in.tum.de/nominal/download.html`.

[29] Christian Urban, James Cheney & Stefan Berghofer (2011): *Mechanizing the metatheory of LF*. ACM Trans. Comput. Logic 12(2), doi:10.1145/1877714.1877721.

[30] Sebastián Urciuoli (2023): *A Formal Proof of the Strong Normalization Theorem for System T in Agda*. In Daniele Nantes-Sobrinho & Pascal Fontaine, editors: *Proc. LSFA '22*, 376, Open Publishing Association, p. 81–99, doi:10.4204/eptcs.376.8.

[31] Sebastián Urciuoli, Álvaro Tasistro & Nora Szasz (2020): *Strong Normalization for the Simply-Typed Lambda Calculus in Constructive Type Theory Using Agda*. In Cláudia Nalon & Giselle Reis, editors: *Proc. LSFA '20*, *ENTCS* 351, Elsevier, pp. 187–203, doi:10.1016/j.entcs.2020.08.010.