Explicit Convertibility Proofs in Pure Type Systems

Floris van Doorn
Utrecht University
florisvandoorn@hotmail.com

Herman Geuvers
Radboud University Nijmegen
herman@cs.ru.nl

Freek Wiedijk
Radboud University Nijmegen
freek@cs.ru.nl

Abstract

We define type theory with explicit conversions. When type checking a term in normal type theory, the system searches for convertibility paths between types. The results of these searches are not stored in the term, and need to be reconstructed every time again. In our system, this information is also represented in the term.

The system we define has the property that the type derivation of a term has exactly the same structure as the term itself. This has the consequence that there exists a natural LF encoding of such a system in which the encoded type is a dependent parameter of the type of the encoded term.

For every Pure Type System we define a system in our style. We show that such a system is always equivalent to the normal system without explicit conversions (even for non-functional systems), in the sense that the typability relation can be lifted. This proof has been fully formalised in the Coq system, building on a formalisation by Vincent Siles.

In our system, explicit conversions are not allowed to be removed when checking for convertibility. This means that all terms in convertibility proofs are well typed, even in the sense of our system.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems

General Terms Reliability, Theory

Keywords Type Theory, Pure Type Systems, Conversion Rule, Convertibility Proofs, Formalisation, Coq

1. Introduction

Dependent type systems are used as a basis for formalising mathematics through the well-known *Curry-Howard formulas-as-types embedding*. Types are used to represent "sets" and "data types," but also to represent "formulas." In that interpretation a proof of a formula A is a term M of type A. So, proofs become first-class citizens of the system and *proof checking* is the same as *type checking*: verifying whether $M : A$ holds. A proof assistant like *Coq* is based on this idea, using a type system called the 'Calculus of Inductive Constructions'. This system also includes a (small) functional programming language: one can define data types as inductive types and program functions over these data types by well-founded recursion.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
LFMTP 2013 23 September 2013, Boston, USA
Copyright © 2013 ACM [to be supplied]. . . $15.00

In dependent type systems, terms occur in types, so one can do (computational) reduction steps inside types. This includes β-reduction steps, but, in case one has inductive types, also ι-reduction and, in case one has definitions, also δ-reduction. A type B that is obtained from A by a (computational) reduction step is considered to be "equal" to A. A common approach to deal with this equality between types is to use an externally defined notion of *conversion*. In case one has only β-reduction, this is β-conversion which is denoted by $A \simeq_\beta B$. This is the least congruence containing the β-reduction step. Then there is a conversion rule of the form

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A' : s}{\Gamma \vdash a : A'} \, (A \simeq_\beta A'). \qquad \text{(conv)}$$

Now, when you apply a term $M : A \to B$ to a term $P : A'$, one needs to check if $A \simeq_\beta A'$. There is a general way to do this: reduce both types to β-normal-form, and check if these normal forms are equal. Unfortunately, reducing a term to β-normal-form might take hyperexponential time. In practice, proof checkers use heuristics to check if two types are β-convertible without reducing to normal form. However, there is in general no efficient way to check β-convertibility.

The phenomenon that β-convertibility – which is a crucial and non-trivial part of the correctness of a proof – is checked by an auxiliary program and not recorded in the proof-term goes against the intuition of what a proof is. A proof should be "self evident" and should encode all information to check its correctness; it should not require additional computation or intelligence to verify a proof. At present, the situation is far from that. Notably in the system Coq, where $\beta\delta\iota$-conversion is used to implement proof search algorithms and automated theorem proving algorithms inside the type system, using the so called *reflection approach*, see for example [8, 13]. The idea is to reflect part of the meta-language in the object language and to write proof search algorithms that Coq needs to execute during the type checking phase. Coq users clearly consider this a feature: use Coq's convertibility check to do proof automation.

So, the conversion rule allows one to trade proving (writing explicit proof terms) for computing (the convertibility check in the type checking algorithm). It is not so clear cut whether this is a good idea. At the TYPES meeting in Kloster Irsee in 1998, Henk Barendregt explained the use of the conversion rule in the reflection method, and Per Martin-Löf seemed to consider this to be a bug, because proofs were not "self explanatory" anymore. Also Dick de Bruijn has often stressed the importance of *weak logical frameworks*, e.g. [6]. He stressed the point that the logical framework should be general and that additional rules for doing logic or computation should be provided by the user.

Contribution In this paper we define the framework PTS_f, which are Pure Type Systems (PTS) with type-conversions explicitly recorded in the proof terms. This implies that type checking is linear in the size of the term and that types are not only determined

up to beta conversion. In particular for functional specifications every term has a unique type. We state and prove the equivalence of the two frameworks: we show that every PTS $\lambda\mathbf{S}$ is equivalent to its PTS_f companion $\lambda_f\mathbf{S}$. The proof proceeds by showing that every PTS_e judgement can be transformed into a PTS_f judgement, where PTS_e are Pure Type Systems with an explicit equality *judgment*, but no equality-proofs recorded in the terms (It has been shown by Siles and Herbelin [15] that PTS_e and PTS are equivalent frameworks). Our equivalence proof involves a number of subtle technical steps, so we have completely formalised this proof in Coq.

Approach The idea is to introduce proof terms of equalities and add these proof terms to the term when the conversion rule is used. So, there is a separate equality judgement of the form $\Gamma \vdash_f H : A = A'$, which means that H codes the proof that A and A' are convertible in context Γ. The conversion rule now becomes

$$\frac{\Gamma \vdash_f a : A \quad \Gamma \vdash_f A' : s \quad \Gamma \vdash_f H : A = A'}{\Gamma \vdash_f a^H : A'}. \qquad \text{(conv)}$$

Note that the convertibility proof H is added to the term. A consequence of this change in the conversion rule is that there is a *unique derivation* of the typing judgement of a term. The equivalence of this formulation of PTS with the original, uses the PTS_e, that uses *typed judgemental equality*. In PTS_e, there are also separate equality judgements, but now of the form $\Gamma \vdash_e A = A' : B$. The difference with PTS_f lies in the fact that

- in PTS_e, the terms A and A' are forced to have the same type,
- in PTS_e, there is no proof term witnessing the equality.

In PTS_e, the conversion rule is

$$\frac{\Gamma \vdash_e a : A \quad \Gamma \vdash_e A = A' : s}{\Gamma \vdash_e a : A'}. \qquad \text{(conv)}$$

We don't prove our equivalence result for one type system, but work in the general setting of Pure Type Systems (PTSs), which allows to build meta-theory for a whole family of type systems. This is to avoid having to build this meta-theory for all type systems one by one. It has already been shown in the literature that a PTS_e is equivalent to the corresponding PTS. In [2] it is shown that these two systems are equivalent for a special family of *functional* PTSs. In [15] this result is generalised to arbitrary PTSs.

Overview of the paper In Section 2 we recall the definition of Pure Type Systems. We assume familiarity with type theory and PTSs, so we don't go into much detail. The article [4] gives a more detailed introduction to Pure Type Systems. As a matter of fact, we do not use the original variant of Pure Types Systems, but an equivalent descriptions (that we have also formally proved to be equivalent in Coq). This version of PTSs uses a judgment to denote that a context is well-formed. The advantage of this version is that then a term uniquely codes for the typing derivation. In Section 3 we describe PTS_es, *Pure Type Systems with typed judgemental equality* $\lambda_e\mathbf{S}$. We state the equivalence between $\lambda\mathbf{S}$ and $\lambda_e\mathbf{S}$.

In Section 4 we introduce PTS_f, *Pure Type Systems with typed convertibility proofs* $\lambda_f\mathbf{S}$. This system is a generalisation of the system λF in paper [7], which is about one particular PTS, λP, the system corresponding to LF. We study the PTS_f λF, which we call $\lambda_f\mathbf{P}$ in this paper more closely in Section 7. We also prove an important property about $\lambda_f\mathbf{S}$, which is that every judgement has a unique derivation.

In Section 5 we prove the equivalence between the PTS $\lambda\mathbf{S}$ and the PTS_f $\lambda_f\mathbf{S}$. In Section 5.1 we study the erasure map in judgements, which is a map from $\lambda_f\mathbf{S}$-terms to $\lambda\mathbf{S}$-terms and we also use this map to prove '$\lambda_f\mathbf{S} \Rightarrow \lambda\mathbf{S}$', which states that a judgement in $\lambda_f\mathbf{S}$ can be transformed to a similar judgement in $\lambda\mathbf{S}$. Then we prove in Section 5.2 the crucial technical result that states that equality is preserved under substitutions. In the final section, 5.3 we prove '$\lambda_e\mathbf{S} \Rightarrow \lambda_f\mathbf{S}$'. Together with the other implication and the equivalence between $\lambda\mathbf{S}$ and $\lambda_e\mathbf{S}$ we conclude that the systems $\lambda\mathbf{S}$ and $\lambda_f\mathbf{S}$ are equivalent. We finish by proving an injectivity statement for products. The proven implications are displayed in Figure 1.1. The equivalence (Theorem 3.1) is the work of Siles.

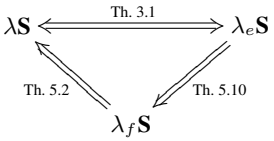

Figure 1.1. Proven implications.

Because the proof of the equivalence between $\lambda\mathbf{S}$ and $\lambda_f\mathbf{S}$ is technical and involves large proofs with many cases, we have also formalised the proof in the proof assistant Coq. In Section 6 we describe this formalisation. Using Coq as proof assistant, we can profit from the fact that Siles has formalised the equivalence between $\lambda\mathbf{S}$ and $\lambda_e\mathbf{S}$ in Coq. We have built the formalisation of our proof on top of the formalisation of Siles, to be able to use his results. The formalisation can be found at `http://www.cs.ru.nl/~freek/ptsf/`. Lemmas and Theorems which have been formalised (mainly in Section 5) state the name of the result in the Coq code using the format `[Coq name]`.

In Section 7 we look more closely at the PTS $\lambda_f\mathbf{P}$ and to a subfamily of PTSs called *functional* PTSs. We prove that in functional PTSs every term has a unique type, and that if we have a convertibility proof between terms, the corresponding types are also convertible. We also present some simplifications to the rules used for PTSs in these particular cases.

2. $\lambda\mathbf{S}$: Pure Type Systems

In this section we introduce the notion of *Pure Type Systems* (PTSs). This is a broad family of type systems, and in this paper will we only treat type systems which can be described as PTS. The *specification* $\mathbf{S}$ of a PTS consists of three sets $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S}$ is the set of *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is the set of *axioms*, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is the set of *relations*. We will use the letters s, t for sorts, possibly adorned with primes or subscripts.

We fix a countably infinite set of variables $\mathcal{V}$. We will denote variables by x, y, z possibly adorned with primes or subscripts. Given a specification $\mathbf{S}$, then we construct the PTS $\lambda\mathbf{S}$ consisting of a set of pseudoterms, pseudocontexts, pseudojudgements and rules to inductively define the judgements. The set $\mathcal{T}$ of *pseudoterms* is constructed using the following *grammar*:

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

We will use the letters $a, b, c, d, A, B, C, D, M, N$, possibly adorned with primes or subscripts, for pseudoterms. Next we define beta reduction. *One step beta reduction* is the compatible closure of the relation

$$(\lambda x{:}A.M)N \rightsquigarrow_\beta M[x := N] \qquad (1)$$

and denoted by $\rightarrow_\beta$. *Beta reduction* is the reflexive transitive closure of one step beta reduction and denoted by $\twoheadrightarrow_\beta$. *Beta conversion* is the reflexive symmetric transitive closure of (one step) beta reduction, i.e. the smallest equivalence relation containing (one step) beta reduction, and denoted by $\simeq_\beta$.

Two important properties about beta conversion are stated in the following theorem. Statement 1 is called confluence or the Church-Rosser theorem.

$$\frac{}{\vdash} \quad \text{(nil)}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash} \; x \notin \operatorname{dom}\Gamma \quad \text{(cons)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash s_1 : s_2} \; (s_1, s_2) \in \mathcal{A} \quad \text{(sort)}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash x : A} \; (x : A) \in \Gamma \quad \text{(var)}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(prod)}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash b : B : s_2}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(abs)}$$

$$\frac{\Gamma \vdash F : \Pi x{:}A.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]} \quad \text{(app)}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A' : s}{\Gamma \vdash a : A'} \; (A \simeq_\beta A') \quad \text{(conv)}$$

Figure 2.1. rules for $\lambda\mathbf{S}$

Theorem 2.1.

1. `[Betas_diamond]` *If for pseudoterms A, B, C we have $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$, then there is a pseudoterm D such that $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.*
2. `[Betac_confl]` *If for pseudoterms A, B we have $A \simeq_\beta B$ then there is a pseudoterm C such that $A \twoheadrightarrow_\beta C$ and $B \twoheadrightarrow_\beta C$.*

Next, we define the set $\mathcal{C}$ of *pseudocontexts* by

$$\mathcal{C} = \cdot \mid \mathcal{C}, \mathcal{V} : \mathcal{T}.$$

Here $\cdot$ is called the *empty context*. Pseudocontexts are denoted by Γ or Δ, possibly adorned with subscripts or primes. All pseudocontexts Γ are of the form (we leave out the dot)

$$\Gamma \equiv x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$$

for some $n \geq 0$. We define $\operatorname{dom}\Gamma = \{x_1, \ldots, x_n\}$ and $(x : A) \in \Gamma$ if $x \equiv x_i$ and $A \equiv A_i$ for some i. We write $\Gamma[x := a]$ for the context

$$x_1 : A_1[x := a], x_2 : A_2[x := a], \ldots, x_n : A_n[x := a].$$

Given two contexts Γ and Δ, we will write the concatenation of these contexts simply by Γ, Δ.

Furthermore we have *pseudojudgements* of the form

$$\mathcal{J} = \mathcal{C} \vdash \mid \mathcal{C} \vdash \mathcal{T} : \mathcal{T}.$$

Here $\Gamma \vdash$ is a *legality* pseudojudgement, meaning that Γ is a legal context and $\Gamma \vdash a : A$ is a *typing* pseudojudgement. We use the abbreviation $\Gamma \vdash A : B : C$ for "$\Gamma \vdash A : B$ and $\Gamma \vdash B : C$."

Alternative rules The rules we gave for judgements is not standard in literature. The most common way to define this, is given in Figure 2.2. Note that in these rules only the typing judgement occurs, and there's no judgement dedicated to stating that a context is legal.

Of course we want to know if the two different presentations of the rules are equivalent.

Proposition 2.2. `[legacy2typ]` `[typ2legacy]` *The rules in Figure 2.1 define the same typing judgements as the rules in Figure 2.2.*

$$\frac{}{\vdash s_1 : s_2} \; (s_1, s_2) \in \mathcal{A} \quad \text{(ax)}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \; x \notin \operatorname{dom}\Gamma \quad \text{(var}')$$

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash b : B}{\Gamma, x : A \vdash b : B} \; x \notin \operatorname{dom}\Gamma \quad \text{(weak)}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(prod)}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash b : B : s_2}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(abs)}$$

$$\frac{\Gamma \vdash F : \Pi x{:}A.B \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]} \quad \text{(app)}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A' : s}{\Gamma \vdash a : A'} \; (A \simeq_\beta A') \quad \text{(conv)}$$

Figure 2.2. alternative rules for $\lambda\mathbf{S}$ more common in literature

The proof is easy by induction on the derivation, we will skip it here.

For a more detailed overview of PTSs, see [4]. In this article, the rules in Figure 2.2 are used, and then the meta-theory of these rules is developed. We will use one nontrivial proposition from this meta-theory.

Theorem 2.3 (Subject Reduction)**.** `[SubjectRed]` *If $\Gamma \vdash A : B$ and $A \twoheadrightarrow_\beta A'$ then $\Gamma \vdash A' : B$.*

Proof. See [4], Lemma 5.2.15 on page 107. □

3. λ_eS: Typed judgemental equality

Given a specification **S**, we define the *Pure Type System with typed judgemental equality* $\lambda_e\mathbf{S}$ as follows. It has the same pseudoterms and pseudocontexts as $\lambda\mathbf{S}$, but there is another kind of pseudojudgement. We will annotate the turnstile with a subscript e to distinguish the judgements in $\lambda_e\mathbf{S}$ from judgements in $\lambda\mathbf{S}$. We still have the ordinary *typing judgement* $\Gamma \vdash_e M : A$ and *legality judgement* $\Gamma \vdash_e$, but we also have an *equality judgement* $\Gamma \vdash_e M = M' : A$. The deduction rules are given in Figure 3.1. The first seven rules are exactly the same, but in the conversion rule we do not use an externally defined beta convertibility anymore, instead, we have to prove the equality within the system. The new rules describe what the equality judgements are. The rules (ref), (sym) and (trans) are for the reflexivity, symmetry and transitivity of the equality. Then (beta) is the analogue of equation (1) in this system, and the rules (prod-eq), (abs-eq) and (app-eq) are to ensure that the equality is compatible with the structure of terms. Finally we also have a conversion rule (conv-eq) for equality judgements

In [2] it is shown that these two different type systems are equivalent for so-called functional specifications (cf. Definition 7.2). In [14] the result is also proven for semi-full specifications. In [15] this equivalence is generalised to arbitrary specifications. The equivalence is formulated as follows.

Theorem 3.1 (Equivalence of $\lambda\mathbf{S}$ and $\lambda_e\mathbf{S}$)**.**

1. $\Gamma \vdash_e$ *iff* $\Gamma \vdash$*;*
2. $\Gamma \vdash_e M : A$ *iff* $\Gamma \vdash M : A$*;*
3. $\Gamma \vdash_e M = N : A$ *iff* $\Gamma \vdash M : A$*,* $\Gamma \vdash N : A$ *and* $M \simeq_\beta N$*.*

The proof is hard and is given in [15].

$$\frac{}{\vdash_e} \quad \text{(nil)}$$

$$\frac{\Gamma \vdash_e A : s}{\Gamma, x : A \vdash_e} \; x \notin \operatorname{dom}\Gamma \quad \text{(cons)}$$

$$\frac{\Gamma \vdash_e}{\Gamma \vdash_e s_1 : s_2} \; (s_1, s_2) \in \mathcal{A} \quad \text{(sort)}$$

$$\frac{\Gamma \vdash_e}{\Gamma \vdash_e x : A} \; (x : A) \in \Gamma \quad \text{(var)}$$

$$\frac{\Gamma \vdash_e A : s_1 \quad \Gamma, x : A \vdash_e B : s_2}{\Gamma \vdash_e \Pi x{:}A.B : s_3} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(prod)}$$

$$\frac{\Gamma \vdash_e A : s_1 \quad \Gamma, x : A \vdash_e b : B : s_2}{\Gamma \vdash_e \lambda x{:}A.b : \Pi x{:}A.B} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(abs)}$$

$$\frac{\Gamma \vdash_e F : \Pi x{:}A.B \quad \Gamma \vdash_e a : A}{\Gamma \vdash_e Fa : B[x := a]} \quad \text{(app)}$$

$$\frac{\Gamma \vdash_e a : A \quad \Gamma \vdash_e A = A' : s}{\Gamma \vdash_e a : A'} \quad \text{(conv)}$$

$$\frac{\Gamma \vdash_e A : B}{\Gamma \vdash_e A = A : B} \quad \text{(ref)}$$

$$\frac{\Gamma \vdash_e A = A' : B}{\Gamma \vdash_e A' = A : B} \quad \text{(sym)}$$

$$\frac{\Gamma \vdash_e A = A' : B \quad \Gamma \vdash_e A' = A'' : B}{\Gamma \vdash_e A = A'' : B} \quad \text{(trans)}$$

$$\frac{\Gamma \vdash_e a : A : s_1 \quad \Gamma, x : A \vdash_e b : B : s_2}{\Gamma \vdash_e (\lambda x{:}A.b)a = b[x := a] : B[x := a]} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(beta)}$$

$$\frac{\Gamma \vdash_e A = A' : s_1 \qquad \Gamma, x : A \vdash_e B = B' : s_2}{\Gamma \vdash_e \Pi x{:}A.B = \Pi x{:}A'.B' : s_3} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(prod-eq)}$$

$$\frac{\Gamma \vdash_e A = A' : s_1 \qquad \Gamma, x : A \vdash_e b = b' : B : s_2}{\Gamma \vdash_e \lambda x{:}A.b = \lambda x{:}A'.b' : \Pi x{:}A.B} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(abs-eq)}$$

$$\frac{\Gamma \vdash_e F = F' : \Pi x{:}A.B \qquad \Gamma \vdash_e a = a' : A}{\Gamma \vdash_e Fa = F'a' : B[x := a]} \quad \text{(app-eq)}$$

$$\frac{\Gamma \vdash_e a = a' : A \quad \Gamma \vdash_e A = A' : s}{\Gamma \vdash_e a = a' : A'} \quad \text{(conv-eq)}$$

Figure 3.1. rules for $\lambda_e\mathbf{S}$

If one tries to prove this directly, then the direction from left to right is easy by induction over the derivation of the judgement, but for the other direction, the equivalence of equality is very hard, as is described in [2]. One could try to derive that if $\Gamma \vdash M : A$ and $M \twoheadrightarrow_\beta N$ then $\Gamma \vdash_e M = N : A$ from which the desired statement follows using Church-Rosser (Theorem 2.1). In normal PTSs, the way to derive such a statement is to prove the following statements simultaneously by induction

- If $\Gamma \vdash_e M : A$ and $M \to_\beta N$ then $\Gamma \vdash_e M = N : A$;
- If $\Gamma \vdash_e M : A$ and $\Gamma \to_\beta \Delta$ then $\Delta \vdash_e M : A$.

Here $x_1 : A_1, \ldots, x_n : A_n \to_\beta x_1 : B_1, \ldots, x_n : B_n$ means that there is a $j \leq n$ such that $A_j \to_\beta B_j$ and that for all $i \neq j$ we have $A_i \equiv B_i$. If one tries to prove this, the hard case is proving (app) for the first statement, specifically if one derived $\Gamma \vdash_e (\lambda x{:}A.b)a : B[x := a]$ with the corresponding reduction $(\lambda x{:}A.b)a \to_\beta b[x := a]$. If one tries to prove $\Gamma \vdash_e (\lambda x{:}A.b)a = b[x := a] : B[x := a]$ then one needs a form of product injectivity, i.e. one needs the following statement. If $\Gamma \vdash_e \Pi x{:}A.B = \Pi x{:}A'.B' : s_3$ then there is a relation $(s_1, s_2, s_3) \in \mathcal{R}$ such that $\Gamma \vdash_e A = A' : s_1$ and $\Gamma, x : A \vdash_e B = B' : s_2$. There is no obvious way to prove this, because the equality could have been derived via a chain of (trans)-rules, and we don't really know much about the terms in the middle of this chain.

The way [15] proved the Theorem was to define a new variant of PTS they called *Pure Type System based on Annotated Typed Reduction* or PTS_{atr}. This system is a typed version of parallel beta reduction [16]. They also needed to add typing information to each application, which means that each application was of the form $M_{\Pi x:A.B}N$ where $\Pi x{:}A.B$ is the type of M. In this system they were able to prove confluence for the typed reduction, and from that they were able to prove a weak form of product injectivity and also subject reduction. Then they proved the equivalence between PTS_{atr} and PTS_e (which we call $\lambda_e\mathbf{S}$). This equivalence implies Theorem 3.1.

4. $\lambda_f\mathbf{S}$: Typed convertibility proofs

For a specification $\mathbf{S}$ we define the *Pure Type System with convertibility proofs* $\lambda_f\mathbf{S}$ as follows. There is a separate class $\mathcal{H}$ of (pseudo-)convertibility proofs and the pseudoterms $\mathcal{T}$ have one extra constructor, the conversion a^H for a pseudoterm a and convertibility proof H.

$$\mathcal{T} = \mathcal{V} \mid \mathcal{S} \mid \mathcal{T}\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\mathcal{T} \mid \mathcal{T}^{\mathcal{H}}.$$

The convertibility proofs have the following grammar, and are denoted by H (possibly adorned with primes or subscripts):

$$\mathcal{H} = \overline{\mathcal{T}} \mid \mathcal{H}^\dagger \mid \mathcal{H} \cdot \mathcal{H} \mid \beta(\mathcal{T}) \mid \{\mathcal{H}, [\mathcal{V} : \mathcal{T}]\mathcal{H}\} \mid \langle \mathcal{H}, [\mathcal{V} : \mathcal{T}]\mathcal{H}\rangle \mid \mathcal{H}\mathcal{H} \mid \iota(\mathcal{H}).$$

Note that the ι in the grammar has nothing to do with ι-reduction. We define $H[x := a]$ in the obvious way, by replacing x with a for every free occurrence of x in H. Note that in $\{H_1, [x : A]H_2\}$ and $\langle H_1, [x : A]H_2\rangle$ the free occurrences of x in H_2 are bound by $[x : A]$ and hence will not be replaced.

The pseudocontexts have the same grammar as before. As in $\lambda_e\mathbf{S}$, there are three different kind of judgements, but the equality judgement is now different. In $\lambda_e\mathbf{S}$, the equality judgement has the form $\Gamma \vdash_e M = N : A$, while in $\lambda_f\mathbf{S}$, the equality judgement has the form $\Gamma \vdash_f H : M = N$. So instead of typing the equality, we have a convertibility proof witnessing the equality. This also means that in $\lambda_f\mathbf{S}$, the terms in an equality judgement a priori need not have the same type, hence this equality is a form of *heterogenous* or *John Major equality* [12]. Also, for non-functional specifications we will see an example of an equality between terms which do not

$$\frac{}{\vdash_f} \quad \text{(nil)}$$

$$\frac{\Gamma \vdash_f A : s}{\Gamma, x : A \vdash_f} \; x \notin \operatorname{dom}\Gamma \quad \text{(cons)}$$

$$\frac{\Gamma \vdash_f}{\Gamma \vdash_f s_1 : s_2} \; (s_1, s_2) \in \mathcal{A} \quad \text{(sort)}$$

$$\frac{\Gamma \vdash_f}{\Gamma \vdash_f x : A} \; (x : A) \in \Gamma \quad \text{(var)}$$

$$\frac{\Gamma \vdash_f A : s_1 \quad \Gamma, x : A \vdash_f B : s_2}{\Gamma \vdash_f \Pi x{:}A.B : s_3} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(prod)}$$

$$\frac{\Gamma \vdash_f A : s_1 \quad \Gamma, x : A \vdash_f b : B : s_2}{\Gamma \vdash_f \lambda x{:}A.b : \Pi x{:}A.B} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(abs)}$$

$$\frac{\Gamma \vdash_f F : \Pi x{:}A.B \quad \Gamma \vdash_f a : A}{\Gamma \vdash_f Fa : B[x := a]} \quad \text{(app)}$$

$$\frac{\Gamma \vdash_f a : A \quad \Gamma \vdash_f A' : s \quad \Gamma \vdash_f H : A = A'}{\Gamma \vdash_f a^H : A'} \quad \text{(conv)}$$

$$\frac{\Gamma \vdash_f A : B}{\Gamma \vdash_f \overline{A} : A = A} \quad \text{(ref)}$$

$$\frac{\Gamma \vdash_f H : A = A'}{\Gamma \vdash_f H^\dagger : A' = A} \quad \text{(sym)}$$

$$\frac{\Gamma \vdash_f H : A = A' \quad \Gamma \vdash_f H' : A' = A''}{\Gamma \vdash_f H \cdot H' : A = A''} \quad \text{(trans)}$$

$$\frac{\Gamma \vdash_f a : A : s_1 \quad \Gamma, x : A \vdash_f b : B : s_2}{\Gamma \vdash_f \beta((\lambda x{:}A.b)a) : (\lambda x{:}A.b)a = b[x := a]} \; (s_1, s_2, s_3) \in \mathcal{R} \quad \text{(beta)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash_f A : s_1 & \Gamma, x : A \vdash_f B : s_2 \\ \Gamma \vdash_f A' : s_1' & \Gamma, x' : A' \vdash_f B' : s_2' \\ \Gamma \vdash_f H : A = A' & \Gamma, x : A \vdash_f H' : B = B'[x' := x^H]\end{array}}{\Gamma \vdash_f \{H, [x : A]H'\} : \Pi x{:}A.B = \Pi x'{:}A'.B'} \quad \text{(prod-eq)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash_f A : s_1 & \Gamma, x : A \vdash_f b : B : s_2 \\ \Gamma \vdash_f A' : s_1' & \Gamma, x' : A' \vdash_f b' : B' : s_2' \\ \Gamma \vdash_f H : A = A' & \Gamma, x : A \vdash_f H' : b = b'[x' := x^H]\end{array}}{\Gamma \vdash_f \langle H, [x : A]H'\rangle : \lambda x{:}A.b = \lambda x'{:}A'.b'} \quad \text{(abs-eq)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash_f F : \Pi x{:}A.B & \Gamma \vdash_f a : A \\ \Gamma \vdash_f F' : \Pi x'{:}A'.B' & \Gamma \vdash_f a' : A' \\ \Gamma \vdash_f H : F = F' & \Gamma \vdash_f H' : a = a'\end{array}}{\Gamma \vdash_f HH' : Fa = F'a'} \quad \text{(app-eq)}$$

$$\frac{\Gamma \vdash_f a : A \quad \Gamma \vdash_f A' : s \quad \Gamma \vdash_f H : A = A'}{\Gamma \vdash_f \iota(a^H) : a = a^H} \quad \text{(iota)}$$

Figure 4.1. rules for $\lambda_f\mathbf{S}$

have the same type in Section 7.1. In summary, the judgements have the grammar (annotating the turnstile with a subscript f)

$$\mathcal{J} = \mathcal{C} \vdash_f \mid \mathcal{C} \vdash_f \mathcal{T} : \mathcal{T} \mid \mathcal{C} \vdash_f \mathcal{H} : \mathcal{T} = \mathcal{T}.$$

The deduction rules are given in Figure 4.1. The first seven rules are exactly the same as before, the conversion rule is different, and the other rules describe how to derive equality judgements. In the conversion rule, the most notable difference is that the convertibility proof H is added to the term, so that you can store exactly which rules were used to derive the equality. Most of the rules for equality judgements correspond to a similar $\lambda_e\mathbf{S}$-rule. There are again rules for reflexivity, symmetry and transitivity. Then we have the beta rule, and rules to equate products, abstractions and applications. In (prod-eq) and (abs-eq) the conditions $(s_1, s_2, s_3), (s_1', s_2', s_3') \in \mathcal{R}$ should also hold. Note that we need much more hypotheses to these rules relative to the rules for PTS_e, because we need both typing information and equality information, and in the rules for PTS_e these could be given in a single judgement. At last we have the (iota)-rule which describes the equality between a term and the same term annotated with a convertibility proof. In Section 7 we look at some special cases for the specification, and will notice that some rules can be simplified in these special cases.

The main motivation for defining the system $\lambda_f\mathbf{S}$ is the following Theorem.

Theorem 4.1. `[unique_der]` *The rules used in the derivation of a judgement are uniquely determined by that judgement.*

Remark 4.2. With "the rules used in the derivation of a judgement J" we mean the *derivation tree* $\operatorname{der}(J)$ (which a priori depends on more than only J) with nodes labeled by (nil), (cons), (sort), ..., (iota), describing which rules are used. For example $\operatorname{der}(\vdash_f)$ is a tree with the single node labeled by (nil), and if we last used the rule (abs)-rule

$$\frac{\Gamma \vdash_f A : s_1 \quad \Gamma, x : A \vdash_f b : B : s_2}{\Gamma \vdash_f \lambda x{:}A.b : \Pi x{:}A.B} \; (s_1, s_2, s_3) \in \mathcal{R}$$

then we have

$$\operatorname{der}(\Gamma \vdash_f \lambda x{:}A.b : \Pi x{:}A.B) = \begin{array}{ccc}\operatorname{der}(\Gamma \vdash_f A : s_1) & \operatorname{der}(\Gamma, x : A \vdash_f b : B) & \operatorname{der}(\Gamma, x : A \vdash_f B : s_2) \\ & \vdots & \\ & \text{(abs)} &\end{array}$$

Using this notation, the statement of Theorem 4.1 becomes that the function der is well-defined from judgements to labeled trees, i.e. it only depends on the judgement. Note that $\operatorname{der}(J)$ does not give all information about the derivation of J. For example, the tree $\begin{array}{c}\text{(nil)} \\ \vdots \\ \text{(sort)}\end{array}$ could have conclusion $\vdash_f s : t$ for all axioms $(s, t) \in \mathcal{A}$. ∅

Before we prove Theorem 4.1 we need some more information. In the (abs)-rule, the term $\lambda x{:}A.b$ in the conclusion does not fully describe the type B of b, while B is required to have a type in the hypotheses. We need to prove that if b has two different types B and B', a derivation of the typing of these types must use the same rules.

Definition 4.3. We call two terms A and B *comparable* if there is an $n \geq 0$ and $n + 2$ terms $A_1, \ldots, A_n, A', B' \in \mathcal{T}$ such that the following statements hold

1. We have $A \equiv \Pi x_1{:}A_1.\Pi x_2{:}A_2.\cdots\Pi x_n{:}A_n.A'$;
2. We have $B \equiv \Pi x_1{:}A_1.\Pi x_2{:}A_2.\cdots\Pi x_n{:}A_n.B'$;
3. Either $A' \equiv B'$ or both A' and B' are sorts.

Note that in particular equal terms are comparable, and that any two sorts are comparable.

Lemma 4.4. `[unique_type_comparable]` *If $\Gamma \vdash_f M : A$ and $\Gamma \vdash_f M : B$ then A and B are comparable.*

Proof. By induction on the structure of M. □

We will prove a little stronger result which implies Theorem 4.1.

Theorem 4.5. `[unique_der_ext]`

1. *Any two derivation trees of $\Gamma \vdash_f$ are equal.*
2. *If M and M' are comparable, then any derivation tree of $\Gamma \vdash_f M : A$ is equal to a derivation tree of $\Gamma \vdash_f M' : A'$.*
3. *Any derivation tree of $\Gamma \vdash_f H : M = N$ is equal to a derivation tree of $\Gamma \vdash_f H : M' = N'$*

Proof. By simultaneous induction on the derivation of the first judgement in each item, distinguishing cases according to the last applied rule. □

We need the following definitions.

Definition 4.6. We define the following concepts for $\lambda_f\mathbf{S}$:

1. Γ is called *legal* (or *well-formed*) if $\Gamma \vdash_f$.
2. M is called a Γ-*term* if there is a judgement with context Γ where M appears in as pseudoterm (outside Γ). This means that either $\Gamma \vdash_f M : A$, $\Gamma \vdash_f N : M$, $\Gamma \vdash_f H : M = N$ or $\Gamma \vdash_f H : N = M$.
3. M is called a *term* iff it is a Γ-term for some context Γ.
4. If $\Gamma \vdash_f M : A$ then M is said to *have a type under* Γ and A is called a Γ-*type*.
5. A is called a Γ-*semitype* iff either A is a sort or $\Gamma \vdash_f A : s$ for some sort s.
6. We define $\Gamma \vdash_f M = N$ to mean there exists an H such that $\Gamma \vdash_f H : M = N$ and in this case we call M and N *convertible*.
7. We define the *erasure* map $|\cdot|$ on pseudoterms by the following recursion:

$$|s| \equiv s \qquad |\Pi x{:}A.B| \equiv \Pi x{:}|A|.|B| \qquad |Fa| \equiv |F||a|$$
$$|x| \equiv x \qquad |\lambda x{:}A.b| \equiv \lambda x{:}|A|.|b| \qquad |a^H| \equiv |a|$$

Thus $|M|$ is the pseudoterm M with all convertibility proofs removed. If M is a term, then $|M|$ need not to be a term, but it always is a $\lambda\mathbf{S}$-term, which we will prove later. We say that M is a *lift* of M' if $|M| \equiv M'$. We extend the erasure map (and the notion of lift) to contexts by

$$|x_1 : A_1, \ldots, x_n : A_n| \equiv x_1 : |A_1|, \ldots, x_n : |A_n|.$$

5. Meta-theory of $\lambda_f\mathbf{S}$

In this section we show the equivalence between the type systems $\lambda_f\mathbf{S}$ and $\lambda\mathbf{S}$ using the equivalence between $\lambda_e\mathbf{S}$ and $\lambda\mathbf{S}$. In Section 5.1 we will prove the implication $\lambda_f\mathbf{S} \to \lambda\mathbf{S}$ and some properties about the erasure map. In Section 5.2 we will prove a Lemma about equality between substitutions. Finally in Section 5.3 we prove the implication $\lambda_e\mathbf{S} \to \lambda_f\mathbf{S}$, and Product Injectivity as Corollary. All results in this section have been formalised in Coq. The name of the corresponding result in Coq is written next to each result.

The system $\lambda_f\mathbf{S}$ satisfies the Weakening, Substitution, Generation and Type Correctness (a Γ-type is a Γ-semitype) properties as usual. We also have the following Lemma.

Lemma 5.1.

1. `[equality_unique]` *If $\Gamma \vdash_f H : A = B$ and $\Gamma \vdash_f H : \tilde{A} = \tilde{B}$, then $A \equiv \tilde{A}$ and $B \equiv \tilde{B}$.*
2. (Equality Typing) `[equality_typing]` *If $\Gamma \vdash_f A = B$, then both A and B have a type under Γ.*

Proof. 1. Induction on the derivation of the first judgement. In (abs-eq) and (prod-eq) one has to note that if $B[x := x^H] \equiv \tilde{B}[x := x^H]$ then $B \equiv \tilde{B}$.
2. Induction on the derivation of the judgement.

□

5.1 Erasure map

In this section we will prove some properties about the erasure map as defined in Definition 4.6.7. We start with the implication $\lambda_f\mathbf{S} \Rightarrow \lambda\mathbf{S}$.

Theorem 5.2. `[PTSF2PTS]`

1. *If $\Gamma \vdash_f$ then $|\Gamma| \vdash$;*
2. *If $\Gamma \vdash_f A : B$ then $|\Gamma| \vdash |A| : |B|$;*
3. *If $\Gamma \vdash_f H : A = B$ then $|A| \simeq_\beta |B|$.*

Proof. We use simultaneous induction on the derivation of the judgement in each statement, distinguishing cases according to the last used rule. All cases are easy. □

Lemma 5.3. *Suppose $\Gamma \vdash_f H : A = A'$ and $\Gamma \vdash_f A : s$. Then*

1. `[subst_wf]` *if $\Gamma, x' : A', \Delta \vdash_f$ then $\Gamma, x : A, \Delta[x' := x^H] \vdash_f$.*
2. `[subst_typ]` *if $\Gamma, x' : A', \Delta \vdash_f M : N$ then $\Gamma, x : A, \Delta[x' := x^H] \vdash_f M[x' := x^H] : N[x' := x^H]$;*
3. `[subst_eq]` *if $\Gamma, x' : A', \Delta \vdash_f H' : M = N$ then $\Gamma, x : A, \Delta[x' := x^H] \vdash_f H'[x' := x^H] : M[x' := x^H] = N[x' := x^H]$;*

Proof. The statements are proved separately from the Substitution Lemma and other Lemmas. □

Proposition 5.4 (Erasure Injectivity)**.** `[erasure_injectivity_term]` *If A and A' have types under Γ and $|A| \equiv |A'|$, then $\Gamma \vdash_f A = A'$.*

Proof. By induction on the structure of A we prove that for all A' and Γ, if A and A' have types under Γ and $|A| \equiv |A'|$, then $\Gamma \vdash_f A = A'$. In every step we use induction on the structure of A'. □

Lemma 5.5.

1. `[erasure_injectivity_term_sort]` *If A has a type under Γ and $|A| \equiv s$ then $\Gamma \vdash_f A = s$.*
2. `[erasure_term]` *If $|A| \equiv |B|$ and $\Gamma \vdash_f a : A$ and B is a Γ-semitype, then there is a lift b of $|a|$ (that is, $|a| \equiv |b|$) such that $\Gamma \vdash_f b : B$.*
3. `[erasure_term_type]` *If $\Gamma \vdash_f a_1 : A_1$ and $\Gamma \vdash_f A_2 : B$ with $|A_1| \equiv |A_2|$ and $|B| \equiv s$, then there is a lift a_3 of $|a_1|$ and a lift A_3 of $|A_1|$ such that $\Gamma \vdash_f a_3 : A_3 : s$.*
4. `[erasure_equality]` *If $\Gamma \vdash_f a_1 = a_2$, $\Gamma \vdash_f a_1 : A$, $\Gamma \vdash_f a_2 : A$, $|A| \equiv |B|$ and B is a Γ-semitype, then there are lifts b_1, b_2 of $|a_1|, |a_2|$ respectively such that $\Gamma \vdash_f b_1 = b_2$, $\Gamma \vdash_f b_1 : B$ and $\Gamma \vdash_f b_2 : B$.*

Proof. The first is by induction on the structure of A. The others follow from previous Lemmas. □

5.2 Equality of substitutions

In the proof of the equivalence between $\lambda_f\mathbf{S}$ and $\lambda_e\mathbf{S}$ we need one more lemma for the (app-eq)-case. If we have a convertibility proof between two applications, concluded by (app-eq), we need to prove that the types are also convertible under the same context. This means that we need to prove Corollary 5.9, but this cannot be done by a simple induction on the first judgement. For this we need a more general statement (Proposition 5.8), where $x : T$ can occur anywhere in the context.

To see why this is the case, let us try to prove Corollary 5.9. So we want to conclude $\Gamma \vdash_f M[x := a_1] = M[x := a_2]$ from the statements

$$\Gamma, x : T \vdash_f M : N;\ \ \Gamma \vdash_f a_1 = a_2;\ \Gamma \vdash_f a_1 : T;\ \ \Gamma \vdash_f a_2 : T.$$

The obvious way to do this is by induction on either M or induction on the derivation of the judgement $\Gamma, x : T \vdash_f M : N$, and these inductions come down to the same thing. So if we do either induction, we have a problem in the product case. Then $\Gamma, x : T \vdash_f \Pi y{:}A.B : s_3$ is concluded from $\Gamma, x : T \vdash_f A : s_1$ and $\Gamma, x : T, y : A \vdash_f B : s_2$. There's no problem with applying the IH to the first judgement, to obtain $\Gamma \vdash_f A[x := a_1] = A[x := a_2]$, but for the second judgement we have a problem. We cannot apply the IH to it, because the declaration $x : T$ does not occur at the end of the context. So we get stuck.

We might now try to prove a similar statement if we replace the first judgement in the assumption with $\Gamma, x : T, y : A \vdash_f M : N$. Of course this will also fail in the product case, because one of the hypotheses to the rule will have an extra declaration to the end of the context. Still, it is illustrative to try this, because it justifies the definition we're about to introduce. Our first question becomes what the exact formulation of the conclusion becomes if we replace the first judgement by $\Gamma, x : T, y : A \vdash_f M : N$. One might guess that the answer is $\Gamma, y : A \vdash_f M[x := a_1] = M[x := a_2]$, but with a little thought one will see this cannot hold in general. The occurrences of y in $M[x := a_1]$ are expected to have type $A[x := a_1]$ instead of A, and similarly, the occurrences of y in $M[x := a_2]$ are expected to have type $A[x := a_1]$. But this gives a problem, because there seems no good context to the judgement $? \vdash_f M[x := a_1] = M[x := a_2]$. By Substitution we know that $M[x := a_1]$ has a type under $\Gamma, y : A[x := a_1]$ and that $M[x := a_2]$ has a type under $\Gamma, y : A[x := a_2]$, but there seems to be no context where both terms have a type, which we need for our equality.

To solve this problem, let's look at what we exactly need in our attempt to prove the product case above. Then we want to apply (prod-eq) to conclude

$$\Gamma \vdash_f \Pi y{:}A[x := a_1].B[x := a_1] = \Pi y{:}A[x := a_2].B[x := a_2].$$

The hypothesis of this rule which is giving trouble is the hypothesis which equates $B[x := a_1]$ and $B[x := a_2]$. The full judgement is

$$\Gamma, y : A[x := a_1] \vdash_f B[x := a_1] = B[x := a_2][y := y^H].$$

Here H is the convertibility proof determined by $\Gamma \vdash_f H : A[x := a_1] = A[x := a_2]$, which we already had by the IH on the first judgement. This gives us exactly the statement we need to prove if the relevant declaration $x : T$ is the second last declaration in the context. Then we need to prove that

$$\Gamma, y : A[x := a_1] \vdash_f M[x := a_1] = M[x := a_2][y := y^H]$$

can be concluded from

$$\Gamma, x : T, y : A \vdash_f M : N;\ \ \Gamma \vdash_f H : A[x := a_1] = A[x := a_2];$$
$$\Gamma \vdash_f a_1 = a_2;\ \ \Gamma \vdash_f a_1 : T;\ \ \Gamma \vdash_f a_2 : T.$$

If we try to prove this with induction, we again fail in the product case, and we need a new Lemma which states what the formulation becomes if we move the relevant declaration $x : T$ to the third last position in the context. In this case we need to prove that

$$\Gamma, y_1 : A_1[x := a_1], y_2 : A_2[x := a_1] \vdash_f$$
$$M[x := a_1] = M[x := a_2][y_1 := y_1^{H_1}][y_2 := y_2^{H_2}]$$

can be concluded from the following six judgements

$$\Gamma, x : T, y_1 : A_1, y_2 : A_2 \vdash_f M : N;\ \ \Gamma \vdash_f a_1 : T;\ \ \Gamma \vdash_f a_2 : T;$$
$$\Gamma \vdash_f a_1 = a_2;\ \ \Gamma \vdash_f H_1 : A_1[x := a_1] = A_1[x := a_2];$$
$$\Gamma, y_1 : A_1[x := a_1] \vdash_f H_2 : A_2[x := a_1] = A_2[x := a_2][y_1 := y_1^H].$$

This illustrates what the general case must be. If the first assumption becomes $\Gamma, x : T, \Delta \vdash_f M : N$ with Δ a context with n declarations, then we need to prove the equality between $M[x := a_1]$ and $M[x := a_2][\cdots]$ where the second term also has n substitutions for all n variables in the domain of Δ. For this we need n equality judgements in our assumptions, proving equalities between the types occurring in Δ with similar substitutions.

Note that a similar problem occurs if one tries to prove the Substitution Lemma where the substituted variable only appears in the last declaration of a judgement. If one tries to prove that $\Gamma, x : A \vdash_f B : C$ and $\Gamma \vdash_f a : A$ implies $\Gamma \vdash_f B[x := a] : C[x := a]$, then one also runs into trouble when doing the product or abstraction case, because a declaration comes after the relevant declaration $x : A$.

Definition 5.6. For a vector $\vec{x} = (x_1, \ldots, x_n)$ write $\vec{x}_i := (x_1, \ldots, x_i)$ and $\vec{x}^i := (x_i, \ldots, x_n)$. We use the same notation for pseudocontexts.

Let $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_n)$ be two vectors of variables, and $\vec{H} = (H_1, \ldots, H_n)$ be a vector of pseudoconvertibility proofs. Define for a pseudoterm M the *n-fold substitution* $M[\vec{x} := \vec{y}^{\vec{H}}] := M[x_1 := y_1^{H_1}][x_2 := y_2^{H_2}] \cdots [x_n := y_n^{H_n}]$. For a context Δ we define $\Delta[\vec{x} := \vec{y}^{\vec{H}}]$ similarly.

First we need some information about typing of these n-fold substitutions. The idea is that we use Lemma 5.3 repeatedly.

Lemma 5.7. *Let Γ and $\Delta = y_1 : D_1, \ldots, y_n : D_n$ and $\Delta' = y_1' : D_1', \ldots, y_n' : D_n'$ be (pseudo)contexts such that Γ, Δ is legal. Suppose for all $i \in \{1, \ldots n\}$ we have $\Gamma, \Delta_{i-1} \vdash_f H_i : D_i = D_i'[\vec{y'}_{i-1} := \vec{y}_{i-1}^{\vec{H}_{i-1}}]$. Then*

1. `[subst_mult_typ]` *If $\Gamma, \Delta' \vdash_f M : N$ then for all $k \leq n$ we have $\Gamma, \Delta_k, \Delta'^{k+1}[\vec{y}_k := \vec{y}_k^{\vec{H}_k}] \vdash_f M[\vec{y}_k := \vec{y}_k^{\vec{H}_k}] : N[\vec{y}_k := \vec{y}_k^{\vec{H}_k}]$;*
2. `[subst_mult_eq]` *If $\Gamma, \Delta' \vdash_f H : M = N$ then for all $k \leq n$ we have $\Gamma, \Delta_k, \Delta'^{k+1}[\vec{y}_k := \vec{y}_k^{\vec{H}_k}] \vdash_f H[\vec{y}_k := \vec{y}_k^{\vec{H}_k}] : M[\vec{y}_k := \vec{y}_k^{\vec{H}_k}] = N[\vec{y}_k := \vec{y}_k^{\vec{H}_k}]$;*

In particular (for $k = n$) we have $\Gamma, \Delta \vdash_f M[\vec{y'} := \vec{y}^{\vec{H}}] : N[\vec{y'} := \vec{y}^{\vec{H}}]$ (and the variant of this for equality judgements).

Proof. The statements are proved separately by induction on k. □

Now we can prove the general statement.

Proposition 5.8. `[equality_subst_ext]` *Let Γ and $\Delta = y_1 : D_1, \ldots, y_n : D_n$ be pseudocontexts. Suppose that $\Gamma \vdash_f a_1 = a_2$ and $\Gamma \vdash_f a_1 : T$ and $\Gamma \vdash_f a_2 : T$ and $\Gamma, x : T, \Delta \vdash_f M : N$ and for all $i \in \{1, \ldots, n\}$ we have $\Gamma, \Delta_{i-1}[x := a_1] \vdash_f H_i : D_i[x := a_1] = D_i[x := a_2][\vec{y}_{i-1} := \vec{y}_{i-1}^{\vec{H}_{i-1}}]$. Then $\Gamma, \Delta[x := a_1] \vdash_f M[x := a_1] = M[x := a_2][\vec{y} := \vec{y}^{\vec{H}}]$.*

Proof. Use induction on the derivation of $\Gamma, x : T, \Delta \vdash_f M : N$. □

Corollary 5.9. [equality_subst] *If $\Gamma, x : T \vdash_f M : N$ and $\Gamma \vdash_f a_1 = a_2$ and $\Gamma \vdash_f a_1 : T$ and $\Gamma \vdash_f a_2 : T$ then $\Gamma \vdash_f M[x := a_1] = M[x := a_2]$.*

Proof. This is the case $n = 0$ of Proposition 5.8. □

5.3 Equivalence

Theorem 5.10. [PTSeq2PTSF] *For all $\lambda_e\mathbf{S}$ contexts Γ, and all $\lambda_e\mathbf{S}$ pseudoterms M, N, T the following statements hold.*

1. *If $\Gamma \vdash_e$ then there is a legal lift Γ' of Γ.*
2. *If $\Gamma \vdash_e M : T$, then there is a legal lift Γ' of Γ, and for every legal lift Γ' of Γ there are lifts M', T' of M, T respectively such that $\Gamma' \vdash_f M' : T'$;*
3. *If $\Gamma \vdash_e M = N : T$, then there is a legal lift Γ' of Γ, and for every legal lift Γ' of Γ there are lifts M', N', T' of M, N, T respectively such that $\Gamma' \vdash_f M' = N'$, $\Gamma' \vdash_f M' : T'$ and $\Gamma' \vdash_f N' : T'$.*

Proof. We prove all statements by simultaneous induction on the derivation of the $\lambda_e\mathbf{S}$-judgement. We distinguish cases according to the last applied rule. □

Theorem 5.11. [PTSl2PTSF] *For all $\lambda\mathbf{S}$-contexts Γ and all $\lambda\mathbf{S}$-terms A and B the following statements hold.*

1. *If Γ is legal then there exists a legal lift of Γ;*
2. *If $\Gamma \vdash A : B$ and Γ' is a legal lift of Γ, then there are lifts A' and B' of A and B respectively such that $\Gamma' \vdash_f A' : B'$;*
3. *If $A \simeq_\beta B$, A and B both have a type under Γ and Γ' is a legal lift of Γ, then there is a convertibility proof H and there are lifts A' and B' of A and B such that $\Gamma' \vdash_f H : A' = B'$.*

Proof. Only the third implication requires some work, because in $\lambda_e\mathbf{S}$ equalities are only allowed between terms with equal types. By the Church-Rosser Theorem there is a term C such that $A \twoheadrightarrow_\beta C$ and $B \twoheadrightarrow_\beta C$. We know that $\Gamma \vdash A : T_1$ and $\Gamma \vdash B : T_2$, so by Subject Reduction we know that $\Gamma \vdash C : T_1$ and $\Gamma \vdash C : T_2$. By Theorem 3.1 we now conclude that $\Gamma \vdash_e A = C : T_1$ and $\Gamma \vdash_e C = B : T_2$. Now by Theorem 5.10 we conclude that there are lifts A', C', C'', B' of A, C, C, B respectively, such that $\Gamma' \vdash_f A' = C'$ and $\Gamma' \vdash_f C'' = B'$ and all four terms have a type under Γ'. By Erasure Injectivity we conclude that $\Gamma' \vdash_f C' = C''$. We conclude that $\Gamma' \vdash_f A' = B'$ using (trans) twice. □

Combining everything, we have proven the equivalence between $\lambda\mathbf{S}$ and $\lambda_f\mathbf{S}$.

Theorem 5.12 (Equivalence between $\lambda\mathbf{S}$ and $\lambda_f\mathbf{S}$)**.** [PTSlequivPTSF] *For all $\lambda\mathbf{S}$-contexts Γ and all $\lambda\mathbf{S}$-terms A and B the following statements hold.*

1. *Γ is legal iff there exists a legal lift of Γ;*
2. *$\Gamma \vdash A : B$ iff there are lifts Γ', A', B' of Γ, A, B respectively such that $\Gamma' \vdash_f A' : B'$;*
3. *$A \simeq_\beta B$ and A and B both have a type under Γ iff there is a convertibility proof H and there are lifts Γ', A', B' of Γ, A, B such that $\Gamma' \vdash_f H : A' = B'$.*

Corollary 5.13 (Product Injectivity)**.** [Prod_Injective] *If $\Gamma \vdash_f \Pi x{:}A.B = \Pi x{:}A'.B'$, then there are convertibility proofs H and H' such that $\Gamma \vdash_f H : A = A'$ and $\Gamma, x : A \vdash_f H' : B = B'[x := x^H]$.*

Proof. By Theorem 5.12 we conclude that $\Pi x{:}|A|.|B| \simeq_\beta \Pi x{:}|A'|.|B'|$. Using Church-Rosser we easily obtain $|A| \simeq_\beta |A'|$ and $|B| \simeq_\beta |B'|$. Also, by Equality Typing (Lemma 5.1.2) both $\Pi x{:}A.B$ and $\Pi x{:}A'.B'$ have a type under Γ, hence the following judgements hold.

$$\Gamma \vdash_f A : s_1, \qquad \Gamma \vdash_f A' : s_1',$$
$$\Gamma, x : A \vdash_f B : s_2, \qquad \Gamma, x : A' \vdash_f B' : s_2'$$

We start with the convertibility between A and A'. By Theorem 5.12 we conclude that $|\Gamma| \vdash |A| : s_1$ and $|\Gamma| \vdash |A'| : s_1'$, hence by Theorem 5.11 there are lifts A_1 and A_1' of $|A|$ and $|A'|$ respectively, such that $\Gamma \vdash_f A_1 = A_1'$. By Equality Typing and Erasure Injectivity (Proposition 5.4), we find that $\Gamma \vdash_f A = A_1$ and $\Gamma \vdash_f A_1' = A'$. By (trans) twice, we find a convertibility proof H such that $\Gamma \vdash H : A = A'$.
The convertibility between B and $B'[x := x^H]$ is proven similarly. This completes the proof. □

6. Formalisation of the proof

The proofs in the previous section are rather technical, and require way more space to write out in full than was available. To give more confidence in the proofs and make sure we did not make any mistakes we have completely formalised all proofs. During the formalisation we also discovered some errors in earlier versions of the proofs of our theorems. We used the proof assistant Coq [11] (version 8.4) for this purpose. As starting point we used the formalisation of Siles [15], who has formalised his proof of Theorem 3.1 in Coq. The most notable difference between the formalisation and this paper is that we used de Bruijn indices [5]. We chose to use these because one of the advantages of de Bruijn indices is that one does not have to consider alpha conversion. There is also a unique way to represent closed terms, and there is a simple *lift operator* [10] (or shift operator [1]) which does not require to check freshness of variables. Siles' formalisation also used de Bruijn indices. We used proper names as variables for this paper because they're easier to read and write.

The Coq files of the formalisation can be found on the web at the address `http://www.cs.ru.nl/~freek/ptsf/`. The files starting with `f_` are a formalisation of the proofs presented in this paper and the rest is Siles' formalisation. The following table is a summary of the files (the number of lines are approximations).

file	description	lines
`f_term`	definition of terms	450
`f_env`	definition of contexts	250
`f_typ`	definition of the type system and some meta-theory	400
`f_typ2`	meta-theory of section 5.1 and 5.2 and uniqueness of derivations	830
`f_equivalence`	the equivalence in section 5.3	300

7. Special cases of PTSs

In this section we will consider the specification **P**, defined below.

Definition 7.1. The specification $\mathbf{P} = (\mathcal{S}_\mathbf{P}, \mathcal{A}_\mathbf{P}, \mathcal{R}_\mathbf{P})$ where

$$\mathcal{S}_\mathbf{P} = \{*, \Box\}; \quad \mathcal{A}_\mathbf{P} = \{(*, \Box)\}; \quad \mathcal{R}_\mathbf{P} = \{(*, *, *), (*, \Box, \Box)\}.$$

This specification is of particular interest, because $\lambda\mathbf{P}$ is closely related to LF Type System [9]. It is also a member of the *lambda cube* [3].

An important property about $\lambda\mathbf{P}$ is that it is functional. We first prove some properties of all functional PTSs.

7.1 Functional PTSs

Definition 7.2. A specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *functional* (or *singly sorted*) if

- For any two axioms $(s_1, s_2), (s_1, s_2') \in \mathcal{A}$ we have $s_2 = s_2'$.
- For any two relations $(s_1, s_2, s_3), (s_1, s_2, s_3') \in \mathcal{R}$ we have $s_3 = s_3'$.

An important property of functional specifications is that every term has a unique type. In $\lambda\mathbf{S}$ this means 'unique up to beta conversion', but in $\lambda_f\mathbf{S}$ this really means unique types (up to alpha conversion).

Lemma 7.3 (Uniqueness of Types)**.** *If $\mathbf{S}$ is functional, $\Gamma \vdash_f M : N$ and $\Gamma \vdash_f M : N'$, then $N \equiv N'$.*

Proof. We prove this by induction on the structure of M. Note that in each case the last step in the derivation of the two judgements $\Gamma \vdash_f M : N$ and $\Gamma \vdash_f M : N'$ is unique in each induction step. □

Proposition 7.4 (Equality between Types)**.** *If $\mathbf{S}$ is functional, $\Gamma \vdash_f M = M'$, $\Gamma \vdash_f M : A$, $\Gamma \vdash_f M' : A'$ then either $\Gamma \vdash_f A = A'$ or $A \equiv A'$ and they're both sorts.*

Proof. We prove this by induction on the derivation of $\Gamma \vdash_f H : M = M'$, distinguishing cases according to the last used rule. □

Remark 7.5. Proposition 7.4 has some interesting consequences for functional type systems. Whenever one uses the rules (prod-eq) or (abs-eq) if $\mathbf{S}$ is functional, then the two relations used are equal. This means that if we replace (prod-eq) and (abs-eq) with the following two rules, the resulting PTS would be equivalent to $\lambda_f\mathbf{S}$. In both rules the condition $(s_1, s_2, s_3) \in \mathcal{R}$ is implied.

$$\frac{\begin{array}{ll}\Gamma \vdash A : s_1 & \Gamma, x : A \vdash B : s_2 \\ \Gamma \vdash A' : s_1 & \Gamma, x' : A' \vdash B' : s_2 \\ \Gamma \vdash H : A = A' & \Gamma, x : A \vdash H' : B = B'[x' := x^H]\end{array}}{\Gamma \vdash \{H, [x : A]H'\} : \Pi x{:}A.B = \Pi x'{:}A'.B'} \text{ (prod-eq}')$$

$$\frac{\begin{array}{ll}\Gamma \vdash_f A : s_1 & \Gamma, x : A \vdash_f b : B : s_2 \\ \Gamma \vdash_f A' : s_1 & \Gamma, x' : A' \vdash_f b' : B' : s_2 \\ \Gamma \vdash_f H : A = A' & \Gamma, x : A \vdash_f H' : b = b'[x' := x^H]\end{array}}{\Gamma \vdash_f \langle H, [x : A]H'\rangle : \lambda x{:}A.b = \lambda x'{:}A'.b'} \text{ (abs-eq}')$$

We will see in Example 7.6 that these rules are more restrictive when the specification is not functional. We will also see in this Example that Proposition 7.4 cannot be generalised to arbitrary type systems, not even when weakened to the statement that both a and b have a type which are equal, i.e. "*If $\Gamma \vdash_f a = b$ then there are terms A and B such that $\Gamma \vdash_f a : A$, $\Gamma \vdash_f b : B$ and either $\Gamma \vdash_f A = B$ or $A \equiv B$.*" ∅

Example 7.6. Given the specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

$$\begin{aligned}\mathcal{S} &= \{*, \Box, \Box', \triangle, \triangle'\};\\ \mathcal{A} &= \{(*, \Box), (*, \Box'), (\Box, \triangle), (\Box', \triangle')\};\\ \mathcal{R} &= \{(\Box, \Box, \Box), (\Box', \Box', \Box')\}.\end{aligned}$$

Now consider $a \equiv \Pi x{:}*.*^{\overline{\Box}}$ and $b \equiv \Pi x{:}*.*^{\overline{\Box'}}$. Note that $\vdash_f *^{\overline{\Box}} : A$ iff $A \equiv \Box$ and that $\vdash_f *^{\overline{\Box'}} : B$ iff $B \equiv \Box'$. Looking which relations could be used to type a and b we can now deduce that $\vdash_f a : A$ iff $A \equiv \Box$ and $\vdash_f b : B$ iff $B \equiv \Box'$. One can derive $\vdash_f \iota\big(*^{\overline{\Box}}\big)^\dagger \cdot \iota\big(*^{\overline{\Box'}}\big) : *^{\overline{\Box}} = *^{\overline{\Box'}}$, and hence by (prod-eq) that

$$\vdash_f \big\{\overline{*}, [x : *]\iota\big(*^{\overline{\Box}}\big)^\dagger \cdot \iota\big(*^{\overline{\Box'}}\big)\big\} : a = b.$$

Also note that $\vdash_f \Box = \Box'$ would imply that $\Box \simeq_\beta \Box'$ which is false by the Church-Rosser Theorem. This tells us two things.

1. Even though $\vdash_f a = b$, there is no type A of a convertible with a type B of b.
2. The rule (prod-eq) can be used where the relations (s_1, s_2, s_3) and (s_1', s_2', s_3') are different relations (and a similar example can be used to show the same for (abs-eq)).

The second remark doesn't necessarily mean we can prove less equalities if we would only allow (prod-eq) with $s_i = s_i'$. Because if we use (prod-eq$'$) twice one can prove

$$\vdash_f \big\{\overline{*}, [x : *]\iota\big(*^{\overline{\Box}}\big)^\dagger\big\} : a = \Pi x{:}*.*$$
$$\vdash_f \big\{\overline{*}, [x : *]\iota\big(*^{\overline{\Box'}}\big)\big\} : \Pi x{:}*.* = b.$$

Then we can still prove $\vdash_f a = b$ using (trans). It is unknown if this trick can be generalised, such that the rules (prod-eq$'$) and (abs-eq$'$) would suffice to prove all convertibilities which are provable in $\lambda_f\mathbf{S}$. ∅

7.2 The system $\lambda_f\mathbf{P}$

In the specification $\mathbf{P}$ one can do another simplification of the rules. In the rules (conv) and (iota) and can leave out the assumption $\Gamma \vdash_f A' : s$ because in this case this is automatically true.

Proposition 7.7. *In $\lambda_f\mathbf{P}$ the following statements hold.*

1. *If $\Gamma \vdash_f H : s = X$ or $\Gamma \vdash_f H : X = s$ then $X \equiv s$.*
2. *If $\Gamma \vdash_f H : A = B$ and $\Gamma \vdash_f A : s$ then $\Gamma \vdash_f B : s$.*

Proof. 1. First note that $s = *$, because s has a type under Γ. Now we prove the statement by induction on H, which uniquely determines the last used rule of the derivation of the judgement in the hypothesis.
2. This follows from Proposition 7.4 and Part 1 of this Proposition. □

Remark 7.8. This means that one can remove the assumption $\Gamma \vdash_f A' : s$ from (conv) and (iota), because if $\Gamma \vdash a : A$ then A is a Γ-semitype by Type Correctness, and since A has a type under Γ by Equality Typing we know that $\Gamma \vdash A : s$ for some sort s.

In other specifications the above Lemma is false, and by removing the assumption $\Gamma \vdash_f A' : s$ from (conv) and/or (iota) either Type Correctness or Equality Typing will fail to be true, as Example 7.10 demonstrates. ∅

Remark 7.9. The system $\lambda_f\mathbf{P}$ has an LF encoding in which the type of the encoded term is a dependent parameter of the LF type encoding the terms. I.e., if

$$\Gamma \vdash_f M : A$$

then in the encoding we have

$$\ulcorner M \urcorner : \mathsf{term}\ \ulcorner A \urcorner$$

For a normal PTS this won't work, as there the type is only determined up to conversion. The LF context for $\lambda_f\mathbf{P}$ is shown in Figure 7.1. Here the simplications in the rules from the previous remark have been applied.

In this encoding there is the subtlety that $*$ both occurs as a *term* of type $\Box$, as well as a *type* (i.e., as the argument of the term type).

To handle this, we have an embedding i from terms to types, and a predicate is_sort that encodes which types are sorts.

We have not proved the adequacy of this encoding. Also we have not yet investigated how an encoding like this can be given for PTS_fs beyond the lambda cube. ∅

Example 7.10. Given the specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

$$\mathcal{S} = \{*, \Box, \triangle\};$$
$$\mathcal{A} = \{(*, \Box), (\Box, \triangle)\};$$
$$\mathcal{R} = \{(\triangle, \triangle, \triangle)\}.$$

Let $\Gamma \equiv A : *, a : A$. Note that Γ is legal, and that by (beta) we can conclude that $\Gamma \vdash_f (\lambda x{:}\Box.*)* : \Box$ and that for $H \equiv \beta((\lambda x{:}\Box.*)*)^\dagger$ we have $\Gamma \vdash_f H : * = (\lambda x{:}\Box.*) * .$ Now $\Gamma \vdash_f A^H : (\lambda x{:}\Box.*)*$. We also have $\Gamma \vdash_f \iota(A^H) : A = A^H$.

Now suppose that we removed the condition $\Gamma \vdash_f A' : s$ from (conv). Then the pseudojudgement $\Gamma \vdash_f a^{\iota(A^H)} : A^H$ would be a valid judgement, and we would have that

$$\Gamma \vdash_f a^{\iota(A^H)} : A^H : (\lambda x{:}\Box.*) * .$$

Also note that A^H does not have a second type (this follows from either Lemma 5.1.1 or Lemma 7.3). This means that Type Correctness would be false. Similarly, if we removed the condition $\Gamma \vdash_f A' : s$ from (iota) but not from (conv), then Equality Typing (Lemma 5.1.2) would be false. Because we really want the properties of Type Correctness and Equality Typing, we have the judgement $\Gamma \vdash_f A' : s$ as hypothesis for the rules (conv) and (iota). ∅

8. Conclusions and Future Research

The fact that a PTS and its companion PTS_f are equivalent amounts to the fact that all conversion information between types, which is implicit in the PTS-term, can be reconstructed to produce a fully annotated PTS_f-term. This PTS_f-term encodes a full typing derivation of the original PTS-term (with all conversion explicitly spelled out), so we can see the proof of Theorem 5.12 as a type-checking algorithm for the PTS. It would be interesting to give this algorithm more explicitly.

The present paper is an extension of [7], where a system λF has been introduced, which is the PTS_f variant of the well-known PTS λP, the type system corresponding to the Edinburgh Logical Framework. The paper [7] left the equivalence of λP and λF as an open problem, but instead proved the equivalence of λP with a system called λH. The system λH also has terms that encode conversions, which are added to the proof terms in the same way as in λF. In λH however, the terms in the equalities do not need to be well-typed in the sense of the system. So in λH, we have $H : A = A'$ if H codes a conversion between the *pseudo-terms* A and A' and the conversion rule is

$$\frac{\Gamma \vdash_h a : A \quad \Gamma \vdash_h A' : s \quad H : A = A'}{\Gamma \vdash_h a^H : A'}. \qquad \text{(conv)}$$

This is closer to the original conversion rule in PTSs, because there also the conversion doesn't have to go through the well-typed terms. An "H-version", $\lambda_h\mathbf{S}$, can be defined for every PTS, and this system is then in between $\lambda\mathbf{S}$ and $\lambda_f\mathbf{S}$: if $\Gamma \vdash_f M : A$, then $\Gamma \vdash_h M : A$, and if $\Gamma \vdash_h M : A$, then $|\Gamma| \vdash |M| : |A|$, so the results that we have proved in the present paper about the relation between $\lambda_f\mathbf{S}$ and $\lambda\mathbf{S}$ immediately extend to $\lambda_h\mathbf{S}$.

As future research, it would be interesting to extend the idea of defining an LF context for arbitrary PTSs. This amounts to parametrizing the definition in Figure 7.1 over a PTS-specification and proving the adequacy of this definition.

$$\begin{array}{rcl}
\mathsf{type} & : & * \\
\mathsf{is_sort} & : & \mathsf{type} \to * \\
\mathsf{term} & : & \mathsf{type} \to * \\
i & : & \Pi s{:}\mathsf{type}.\ \mathsf{is_sort}\ s \to \mathsf{term}\ s \to \mathsf{type} \\
\mathsf{eq} & : & \Pi A,B{:}\mathsf{type}.\ \mathsf{term}\ A \to \mathsf{term}\ B \to * \\
\mathsf{ref} & : & \Pi A{:}\mathsf{type}.\ \Pi a{:}\mathsf{term}\ A.\ \mathsf{eq}\ AAaa \\
\mathsf{sym} & : & \Pi A,B{:}\mathsf{type}.\ \Pi a{:}\mathsf{term}\ A.\ \Pi b{:}\mathsf{term}\ B. \\
 & & \quad \mathsf{eq}\ ABab \to \mathsf{eq}\ BAba \\
\mathsf{trans} & : & \Pi A,B,C{:}\mathsf{type}.\ \Pi a{:}\mathsf{term}\ A.\ \Pi b{:}\mathsf{term}\ B.\ \Pi c{:}\mathsf{term}\ C. \\
 & & \quad \mathsf{eq}\ ABab \to \mathsf{eq}\ BCbc \to \mathsf{eq}\ ACac \\
\mathsf{conv} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A,A'{:}\mathsf{term}\ s. \\
 & & \quad \Pi a{:}\mathsf{term}(isSA).\ \mathsf{eq}\ ssAA' \to \mathsf{term}(isSA') \\
\mathsf{iota} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A,A'{:}\mathsf{term}\ s. \\
 & & \quad \Pi a{:}\mathsf{term}(isSA).\ \Pi H{:}\mathsf{eq}\ ssAA'. \\
 & & \quad \mathsf{eq}(isSA)(isSA')a(\mathsf{conv}\ sSAA'aH) \\
\mathsf{box} & : & \mathsf{type} \\
\mathsf{box_sort} & : & \mathsf{is_sort}\ \mathsf{box} \\
\mathsf{star}' & : & \mathsf{term}\ \mathsf{box} \\
\mathsf{star} & : & \mathsf{type}\ :=\ i\ \mathsf{box}\ \mathsf{box_sort}\ \mathsf{star} \\
\mathsf{star_sort} & : & \mathsf{is_sort}\ \mathsf{star}' \\
i_* & : & \mathsf{term}\ \mathsf{star} \to \mathsf{type}\ :=\ \lambda A{:}\mathsf{term}\ \mathsf{star}.\ i\ \mathsf{star}\ \mathsf{star_sort}\ A \\
\mathsf{conv}_* & : & \Pi A,A'{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \mathsf{term}(i_*A) \to \mathsf{eq}\ \mathsf{star}\ \mathsf{star}\ AA' \to \mathsf{term}(i_*A') \\
 & & := \lambda A,A'{:}\mathsf{term}\ \mathsf{star}.\ \lambda a{:}\mathsf{term}(i_*A). \\
 & & \quad \lambda H{:}\mathsf{eq}\ \mathsf{star}\ \mathsf{star}\ AA'.\ \mathsf{conv}\ \mathsf{star}\ \mathsf{star_sort}\ AA'aH \\
\mathsf{prod} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \Pi B{:}(\mathsf{term}(i_*A) \to \mathsf{term}\ s).\ \mathsf{term}\ s \\
\mathsf{abs} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \Pi B{:}(\mathsf{term}(i_*A) \to \mathsf{term}\ s). \\
 & & \quad (\Pi x{:}\mathsf{term}(i_*A).\ \mathsf{term}(isS(Bx))) \to \\
 & & \quad \mathsf{term}(isS(\mathsf{prod}\ sSAB)) \\
\mathsf{app} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \Pi B{:}(\mathsf{term}(i_*A) \to \mathsf{term}\ s). \\
 & & \quad \mathsf{term}(isS(\mathsf{prod}\ sSAB)) \to \Pi a{:}(\mathsf{term}(i_*A)). \\
 & & \quad \mathsf{term}(isS(Ba)) \\
\mathsf{beta} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \Pi B{:}(\mathsf{term}(i_*A) \to \mathsf{term}\ s). \\
 & & \quad \Pi b{:}(\Pi x{:}\mathsf{term}(i_*A).\ \mathsf{term}(isS(Bx))). \\
 & & \quad \Pi a{:}\mathsf{term}(i_*A). \\
 & & \quad \mathsf{eq}(isS(Ba))(isS(Ba)) \\
 & & \qquad (\mathsf{app}\ sSAB(\mathsf{abs}\ sSABb)a)(ba) \\
\mathsf{prod_eq} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A,A'{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \Pi B{:}(\mathsf{term}(i_*A) \to \mathsf{term}\ s). \\
 & & \quad \Pi B'{:}(\mathsf{term}(i_*A') \to \mathsf{term}\ s). \\
 & & \quad \Pi H{:}\mathsf{eq}\ \mathsf{star}\ \mathsf{star} AA'. \\
 & & \quad (\Pi x{:}\mathsf{term}(i_*A).\ \mathsf{eq}\ ss(Bx)(B'(\mathsf{conv}_*AA'xH))) \to \\
 & & \quad \mathsf{eq}\ ss(\mathsf{prod}\ sSAB)(\mathsf{prod}\ sSA'B') \\
\mathsf{abs_eq} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A,A'{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \Pi B{:}(\mathsf{term}(i_*A) \to \mathsf{term}\ s). \\
 & & \quad \Pi B'{:}(\mathsf{term}(i_*A') \to \mathsf{term}\ s). \\
 & & \quad \Pi b{:}(\Pi x{:}\mathsf{term}(i_*A).\ \mathsf{term}(isS(Bx))). \\
 & & \quad \Pi b'{:}(\Pi x{:}\mathsf{term}(i_*A').\ \mathsf{term}(isS(B'x))). \\
 & & \quad \Pi H{:}\mathsf{eq}\ \mathsf{star}\ \mathsf{star} AA'. \\
 & & \quad (\Pi x{:}\mathsf{term}(i_*A). \\
 & & \qquad \mathsf{eq}\ (isS(Bx))(isS(B'(\mathsf{conv}_*AA'xH))) \\
 & & \qquad\quad (bx)(b'(\mathsf{conv}_*AA'xH))) \to \\
 & & \quad \mathsf{eq}\ (isS(\mathsf{prod}\ sSAB))(isS(\mathsf{prod}\ sSA'B')) \\
 & & \qquad (\mathsf{abs}\ sSABb)(\mathsf{abs}\ sSA'B'b') \\
\mathsf{app_eq} & : & \Pi s{:}\mathsf{type}.\ \Pi S{:}\mathsf{is_sort}\ s.\ \Pi A,A'{:}\mathsf{term}\ \mathsf{star}. \\
 & & \quad \Pi B{:}(\mathsf{term}(i_*A) \to \mathsf{term}\ s). \\
 & & \quad \Pi B'{:}(\mathsf{term}(i_*A') \to \mathsf{term}\ s). \\
 & & \quad \Pi F{:}(\mathsf{term}(isS(\mathsf{prod}\ sSAB))). \\
 & & \quad \Pi F'{:}(\mathsf{term}(isS(\mathsf{prod}\ sSA'B'))). \\
 & & \quad \Pi a{:}(\mathsf{term}(i_*A)).\ \Pi a'{:}(\mathsf{term}(i_*A')). \\
 & & \quad \mathsf{eq}(isS(\mathsf{prod}\ sSAB))(isS(\mathsf{prod}\ sSA'B'))FF' \to \\
 & & \quad \mathsf{eq}(i_*A)(i_*A')aa' \to \\
 & & \quad \mathsf{eq}(isS(Ba))(isS(B'a')) \\
 & & \qquad (\mathsf{app}\ sSABFa)(\mathsf{app}\ sSA'B'F'a')
\end{array}$$

Figure 7.1. LF context for $\lambda_f\mathbf{P}$

Another interesting issue is to extend this work with δ-reductions (for unfolding definitions) and ι-reductions (for well-founded recursion over inductive types). A practical implementation of type theory has definitions, and many of them also have inductive types.

Finally, for PTS_fs we have come across the interesting problem whether the system with the rules (prod-eq) and (abs-eq) replaced by (prod-eq$'$) and (abs-eq$'$), as given on page 9, is equivalent to the original one. In Section 7.1 we have proven that this is the case for all functional PTSs, but for non-functional PTSs it is open.

Acknowledgments

Thanks to James McKinna and Randy Pollack for valuable discussions on the work described in this paper. Thanks to Vincent Siles for allowing us to use his formalisation as the base for our formalisation.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lvy. Explicit substitutions, 1996.

[2] R. Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, Mar. 2006.

[3] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[4] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[5] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[6] N. G. de Bruijn. A plea for weaker frameworks. In G. Huet and G. Plotkin, editors, *Logical frameworks*, pages 40–67. Cambridge University Press, New York, NY, USA, 1991. ISBN 0-521-41300-1. URL `http://dl.acm.org/citation.cfm?id=120477.120479`.

[7] H. Geuvers and F. Wiedijk. A logical framework with explicit conversions. *Electronic Notes in Theoretical Computer Science*, 199(0):33 – 47, 2008.

[8] H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational reasoning via partial reflection. In M. Aagaard and J. Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 162–178. Springer, 2000. ISBN 3-540-67863-8.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, Jan. 1993.

[10] G. Huet. Constructive computation theory. *Course notes on lambda calculus, University of Bordeaux I*, 2011. URL `http://pauillac.inria.fr/~huet/CCT/`.

[11] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. URL `http://coq.inria.fr`. Version 8.4.

[12] C. McBride. Elimination with a motive. In *Types for Proofs and Programs (Proceedings of the International Workshop, TYPES00), volume 2277 of LNCS*, pages 197–216. Springer-Verlag, 2002.

[13] M. Oostdijk and H. Geuvers. Proof by computation in the coq system. *Theor. Comput. Sci.*, 272(1-2):293–314, 2002.

[14] V. Siles and H. Herbelin. Equality is typable in semi-full pure type systems. In *Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science*, LICS '10, pages 21–30. IEEE Computer Society, 2010.

[15] V. Siles and H. Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(2):153 – 180, May 2012.

[16] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120 – 127, 1995.