Master thesis
Explicit convertibility proofs in Pure Type Systems

Floris van Doorn, Utrecht University

July 2, 2013

Supervisors: Freek Wiedijk, Radboud University Nijmegen
Jaap van Oosten, Utrecht University
Benno van den Berg, Utrecht University

Contents

$$\begin{tikzcd}
& \lambda\omega \arrow[rr] & & \lambda C \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
& \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 1.1: The lambda cube.

1 Introduction

In the last century there has been a lot of development in the foundations of mathematics. A main branch is set theory, which is widely considered as the foundation of mathematics. But there are other systems, one of which is type theory.

1.1 Type Theory

There are several different versions of type theory. The *simply typed lambda calculus* was introduced in [Church, 1940], which is a very basic type theory. Later *system F* was discovered independently in [Girard, 1972] and in [Reynolds, 1974] and the *Calculus of Constructions* was introduced in [Coquand and Huet, 1986]. The type theories system F and Calculus of Constructions are notable for being impredicative, which means that the definition of an object can invoke that object. A famous example of impredicativity is Russell's paradox, where a set containing all sets which do not contain itself is defined. These type theories are consistent because they are constructive (almost all type theories are constructive). Another type theory is the Martin-Löf type theory [Martin-Löf, 1984], which is notable for having *inductive types*, which means that types with an inductive structure are defined, like the natural numbers.

The lambda cube [Barendregt, 1991] consists of eight different type theories, including the three systems mentioned before. The lambda cube is shown in Figure 1.1; the names $\lambda{\to}$, $\lambda 2$ and λC correspond to simply typed lambda calculus, System F and the calculus of constructions, respectively. The arrows are inclusions. We will focus on the system λP, which is a simple system with dependent products (see below). This type system is closely related to the Edinburgh Logical Framework called LF [Harper et al., 1993].

All systems in the lambda cube, and many more, can be described in the general framework of *Pure Type Systems* (PTSs), and we will use this framework in this thesis.

The language of type theories consist of *terms* and *types*.[1] Terms have a type, or are "in a type," so one can think of a type as a set in set theory. A *judgement* $\Gamma \vdash M : A$ means that the term M has type A in *context* Γ. A context is a list of *declarations* $y : A_i$ stating that the variable y has type A_i. An example of a (simple) judgement is $y : A \vdash y : A$. This judgement simply states that if y has type A then y has type A.

Type theories are interesting because of the *Curry-Howard isomorphism* [Howard, 1980]. This states that a variety of logical systems using natural deduction can be encoded in versions of type theories. Under this isomorphism, propositions in the logical system are identified with types in the type theory. Proofs of a proposition are identified with terms having the corresponding type. So under the isomorphism finding a proof of a proposition becomes finding an inhabitant

[1]Actually, in PTSs we do not distinguish terms from types as syntactic categories, but we will ignore this fact for the moment.

of a type. Suppose that we have deduced in the logical system that B follows from $A_1, \ldots, A_n$. We can write this as $\Gamma \vdash_\ell B$, where Γ is the list $A_1, \ldots, A_n$, using the subscript ℓ to distinguish it from a typing judgement. The Curry-Howard isomorphism now says that $\overline{\Gamma} \vdash M : \overline{B}$, where M is an encoding of the deduction and where $\overline{\Gamma}$ and $\overline{B}$ translate the logic to type theory. For example in a simple deduction system we can derive $A \to (A \to B) \to B$, which should be read as $A \to ((A \to B) \to B)$. We can give the following proof using *natural deduction*, using implication elimination ($\to_e$) and implication introduction ($\to_i$):

$$\cfrac{\cfrac{\cfrac{A, A \to B \vdash_\ell A \to B \qquad A, A \to B \vdash_\ell A}{A, A \to B \vdash_\ell B}\to_e}{A \vdash_\ell (A \to B) \to B}\to_i}{\vdash_\ell A \to (A \to B) \to B}\to_i$$

Now under the Curry-Howard isomorphism this becomes

$$A : *, B : * \vdash \lambda x{:}A.\lambda y{:}(A \to B).yx : A \to (A \to B) \to B.$$

As you can see the proposition in the logic has the exact same notation as the type in the judgement. In the judgement there are several new notations, the *function space* $X \to Y$, the *abstraction* $\lambda x{:}A.M$ (which could be considered as the function $x \mapsto M$ with domain A, where x can occur in M) and the *application* MN which is the output of the function M on input N. The context $A : *, B : *$ states that A and B are types. Also note that the term $\lambda x{:}A.\lambda y{:}(A \to B).yx$ in the judgement encodes the used inference rules in the proof given above. Here we read the term from left to right and the proof bottom-up. In that case abstraction corresponds to implication introduction and application corresponds to implication elimination. The variables x and y refer to the assumptions. One can extend the type theory to also include conjunction, disjunction, negation, universal quantification and existential quantification.

If we want to have a type system which corresponds to a logic with universal quantification we need *dependent products* of the form $\Pi x{:}A.B$, where x can occur in B. These can be considered as space of functions $x \mapsto b$ where b has type B (both b and B can depend on x). Dependent products are a generalisation of function spaces. If B does not contain x, then $\Pi x{:}A.B$ is equal to $A \to B$. As example, suppose that we have a proposition P on natural numbers and another proposition Q. Then we can prove

$$\vdash_\ell (Q \to \forall n \in \mathbb{N}.P(n)) \to \forall n \in \mathbb{N}.(Q \to P(n)).$$

In type theory this becomes:

$$\begin{aligned} \mathbb{N} : *, P : \mathbb{N} \to *, Q : * \ \vdash\ & \lambda x{:}(Q \to \Pi n : \mathbb{N}.Pn).\lambda n{:}\mathbb{N}.\lambda y{:}Q.xyn \ : \\ & (Q \to \Pi n : \mathbb{N}.Pn) \to \Pi n{:}\mathbb{N}.(Q \to Pn). \end{aligned}$$

The term again corresponds to a derivation of the statement. Again, abstractions correspond to introductions (of either implications or universal quantifications), applications correspond to eliminations and variables denote assumptions.

Type theories have been used as a basis for proof assistants. A proof assistant is a computer program which helps a user formalising and checking proofs of propositions. In some proofs there are a lot of cases to be checked or the proof is very hard and long. In this case a proof assistant might help improve the confidence in the correctness of a proof. Also, if the proof contains small errors a proof assistant will find them because of the big amount of details one has to add.

We give two examples where using a proof assistant for a proof can be considered useful. An example in computer science is proving the correctness of the compiler of the programming language C. This has been formalised, because a lot of cases need to be checked [Blazy et al., 2006]. An example in mathematics is the proof of the Four Colour Theorem, which reduces the problem to hundreds of different cases which have been proven by computer algorithms. Some mathematicians feel uncomfortable with this proof, because these computer algorithms might contain mistakes.

Gonthier has formalised this proof in the proof assistant Coq [Gonthier, 2005]. Coq is a proof assistant which works in the type theory *predicative Calculus of Inductive Constructions* (pCiC) which itself is based on the Calculus of Constructions λC. This formalisation makes the proof more trustworthy. If one trusts the kernel of Coq and one verifies that the Theorem which Gonthier has formalised is indeed the Four Colour Theorem, then one should be convinced that the Four Colour Theorem is true. Now trusting the kernel of Coq might be equally bad as trusting the code checking the different cases, but there are actually good reasons why one can trust the kernel of Coq [Geuvers, 2009].

1.2 Conversion rule

In Pure Type Systems the judgements are generated by a set of rules, which we will introduce in Chapter 2. One of these rules is the conversion rule. This rule loosely states that if a term M has type A and A is "computationally equal" to B then M has type B. There are different notions of computational equality used for the conversion rule. One widely used notion for equality in the conversion rule is the externally defined notion of *beta convertibility*. Another notion which is used is *typed judgemental equality*, leading to a variant of Pure Type Systems called PTS$_e$. The equivalence of these two equalities has been an open problem for quite a while, but was solved in [Siles and Herbelin, 2012].

Let's consider the following example to see why the conversion rule is needed. Suppose we have a set A, a proposition P on A and we want to prove the second order statement

$$(\forall f \in A^A.\forall a \in A.P(f(a))) \to \forall a \in A.P(a).$$

This statement is easily proven, just take the identity function for f. If we use the Curry-Howard isomorphism on this statement, we want something like (denoting $B = \Pi f : A \to A.\Pi a : A.P(f\,a))$

$$A : *, P : A \to * \vdash \lambda M{:}B.\lambda a{:}A.M(\lambda b{:}A.b)a : B \to \Pi a{:}A.Pa.$$

(note that $\lambda b{:}A.b$ is the identity function on A) which can be obtained from

$$A : *, P : A \to *, M : B, a : A \vdash M(\lambda b{:}A.b)a : Pa \tag{1}$$

Unfortunately, judgement (1) is not quite what we get without the conversion rule. We know that M has type B, so the term Mg has type $\Pi a : A.P(g\,a)$. This means that $M(\lambda b{:}A.b)a$ has type $P((\lambda b{:}A.b)a)$ instead of Pa. These two types are not definitionally equal, but they are beta convertible, so we can use the conversion rule. Then we get that $M(\lambda b{:}A.b)a$ (also) has type Pa, which implies that the judgement (1) is valid, but we need the conversion rule to conclude it.

The conversion rule using beta conversion is usually defined in the following way

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

Here the judgements above the line are assumptions, and the condition to the right of the line is an assumption which is not a judgement. The conclusion is below the line. In the rule a, A and A' range over all possible terms, and s ranges over a set called *sorts*. The judgement $\Gamma \vdash A' : s$ states that A' is a type (compare with $*$ used above; in the systems of the lambda cube $*$ is a sort). The side condition $A \simeq_\beta A'$ means that A and A' are beta convertible. So in words the conversion rule states that if a term a has a type A, and A is beta convertible to the type A', then a also has type A'.

This conversion rule follows the *Poincaré principle*, which states that there is a distinction between computations and proof and that computations do not require a proof [Poincaré, 1905]. This principle sounds reasonable: In the above example we had to establish $P((\lambda b{:}A.b)a) \simeq_\beta Pa$, which is trivially true. We don't need a proof for such trivial things. But following the Poincaré principle also has disadvantages. Suppose we have a very long and complicated computation, is it really satisfactory if we completely skip the computation, and just state that the equality holds?

An even more troubling example is the following. We want to find a proof of a statement P. Suppose our type system is expressive enough to write an algorithm that checks all possible terms by increasing length and then checks whether that term has type P. Then running this algorithm is one big computation. Moreover, if P is indeed inhabited, then one can run this algorithm, and after a (long) while one finds an inhabitant. If we can leave out this whole computation in the proof, then we find a very short proof of the statement P, because the biggest part is left out. So can we really accept such a thing as a proof of our statement P?

1.3 The system PTS$_f$

The conversion rule follows the Poincaré principle, because there is no trace of the computations done for the equality $A \simeq_\beta A'$ in the term (=proof) a. In this thesis we will describe an alternative of PTS, called PTS$_f$, the *Pure Type System with typed convertibility proofs.*[2] We will write the turnstile ($\vdash$) in judgements in this system with a subscript f to distinguish them from PTS judgements. So a typing judgement becomes $\Gamma \vdash_f M : A$. In PTS$_f$ there will be a "convertibility proof" as a witness of the equality between two terms. There will also be a separate equality judgement of the form $\Gamma \vdash_f H : A = A'$, which means that H is the convertibility proof witnessing that A is (computationally) equal to A' in context Γ. In this system the conversion rule is

$$\frac{\Gamma \vdash_f a : A \quad \Gamma \vdash_f A' : s \quad \Gamma \vdash_f H : A = A'}{\Gamma \vdash_f a^H : A'}. \tag{conv}$$

There are some important differences between the conversion rule in PTS and the conversion rule in PTS$_f$. First, in a PTS the terms A and A' should be beta convertible, while in PTS$_f$ the terms should be proven equal in the current context by some proof H. Second, in PTS$_f$ the proof H of the equality is added to the term a after conversion. This means that if we get a proof (term) a of proposition (type) A, we can check all uses of the conversion used in the derivation, because all steps for these convertibilities are in the term a.

The system PTS$_f$ has some interesting properties. Given a valid judgement, there is a unique way to derive this judgement.[3] This means that the term is the complete proof of the type. An other important property is that the type of a term is not only determined up to beta conversion. If a term M has type $B[x := a]$, then it does *not* also have type $(\lambda x : A.B)a$. In particular, for functional specifications (see Section 7.1) every term has a unique type. An important question is whether the systems PTS and PTS$_f$ are equivalent. This equivalence is the main result of this thesis. We prove that one can add convertibility proofs to a PTS judgement to obtain a (valid) PTS$_f$ judgement, and the other way around, if one removes the convertibility proofs from a PTS$_f$ judgement one obtains a (valid) PTS judgement.

To prove the equivalence between PTS and PTS$_f$ we will use the system PTS$_e$. This type system uses *typed judgemental equality.* This means that there are also separate equality judgements, which are of the form $\Gamma \vdash_e A = A' : B$. This is different from the equality judgements in PTS$_f$, for two reasons. First, the terms A and A' are forced to have the same type, and second, there is no proof term witnessing the equality. In this case the conversion rule is

$$\frac{\Gamma \vdash_e a : A \quad \Gamma \vdash_e A = A' : s}{\Gamma \vdash_e a : A'}. \tag{conv}$$

This system combines ideas from Martin-Löf type theory with Pure Type Systems. In Martin-Löf type theory there are also equality judgements, and in these equality judgements the terms being equal are forced to have the same type.

[2]The f stands for "fully well-typed." This name is from [Geuvers and Wiedijk, 2008], which is mainly about λH, an untyped version of PTS$_f$. In the discussion the system λF was introduced as an typed alternative to λH, and λF is be the λP-version of PTS$_f$.

[3]What we mean with this is that if one writes the derivation as a tree, where the nodes are labelled with the used rule, then this *derivation tree* is unique. For non-functional PTSs this does not completely determine the derivation.

$$\begin{tikzcd}
\mathrm{PTS} \arrow[rr, Leftrightarrow, "\text{Th. 3.1 (Siles)}"] & & \mathrm{PTS}_e \arrow[dl, Rightarrow, "\text{Th. 5.21}"] \\
& \mathrm{PTS}_f \arrow[ul, Rightarrow, "\text{Th. 5.13}"] &
\end{tikzcd}$$

Figure 1.2: Proven implications.

In [Adams, 2006] it is shown that the systems PTS and PTS_e are equivalent for a special family of *functional* PTSs. In [Siles and Herbelin, 2012] this result is generalised to arbitrary PTSs. We use the result of Siles to prove the equivalence of PTS and PTS_f. In particular, we will prove the implications[4] $\mathrm{PTS}_f \Rightarrow \mathrm{PTS}$ and $\mathrm{PTS}_e \Rightarrow \mathrm{PTS}_f$ in this paper, and using $\mathrm{PTS} \iff \mathrm{PTS}_e$, we conclude $\mathrm{PTS} \iff \mathrm{PTS}_f$. A diagram of these implications can be found in Figure 1.2.

We have completely formalised the equivalence of PTS and PTS_f in the proof assistant Coq. We did this for two reasons. One reason was to ensure there were no errors in the proofs, which was hard to check on paper, because the proofs have a lot of case distinctions. In fact, during the formalisation we did find a few mistakes in the proof, some of which were nontrivial to correct. The second reason was to investigate the use of proof assistants in mathematics. Right now proof assistants are only used by specialists, because one needs to add a lot of details to a proof before the proof assistant accepts it as a valid proof. But in the future proof assistants will become more and more accessible, and maybe there will become a time where regular mathematicians will use proof assistants. We have built our formalisation on top of the formalisation of Siles, which was also in Coq.

1.4 Overview of this thesis

In this thesis we work with different versions of the framework of Pure Type Systems. If we have a variant, we will denote PTS_x for the family of Pure Type Systems and $\lambda_x\mathbf{S}$ for a member of that family, with specification $\mathbf{S}$. An outline of the thesis is the following:

In chapter 2 we will define Pure Type Systems PTS. This chapter will not assume any foreknowledge about Pure Type Systems, although it is not feasible to go into great detail in this chapter. Audience which is new to the subject of type systems might want to consult additional literature. The book [Barendregt, 1992] is an excellent and detailed description of Pure Type Systems. Actually we will present two very similar descriptions of PTSs, and we will prove the equivalence between the two descriptions.

In chapter 3 we will describe the theory of Pure Type Systems with typed judgemental equality PTS_e. We will also state the equivalence between PTS and PTS_e and give an outline of the proof of it, found in [Siles and Herbelin, 2012].

In chapter 4 we will introduce Pure Type Systems with typed convertibility proofs PTS_f. This system is a generalisation of the system λF defined in the discussion of the paper [Geuvers and Wiedijk, 2008]. The system λF is one particular PTS, corresponding to the system λP of the lambda cube.

In chapter 5 we will prove the equivalence between the Pure Type Systems PTS and PTS_f. We will first prove in Section 5.1 some basic properties about variables. In Section 5.2 we will prove some basic properties of the meta-theory of PTS_f, which is similar to the basic meta-theory of ordinary PTSs. In section 5.3 we prove an important property about PTS_f, which is that every judgement has a unique derivation. In section 5.4 we will study the erasure map in judgements, which is a map from PTS_f-terms to PTS-terms and we also use this map to prove '$\mathrm{PTS}_f \Rightarrow \mathrm{PTS}$', which states that a judgement in PTS_f can be transformed to a similar judgement in PTS. Then

[4]They're not actually implications, because we're not talking about propositions. What we mean with for example $\mathrm{PTS}_f \Rightarrow \mathrm{PTS}$ is that if we have a PTS_f-judgement, then we can transform it to a valid PTS-judgement. For the precise formulations, see the Theorems referenced in Figure 1.2.

we prove a result which states that equality is preserved under substitutions in section 5.5. In the final section, 5.6 we will prove '$\text{PTS}_e \Rightarrow \text{PTS}_f$'. Together with the other implication and the equivalence between PTS and PTS_e we conclude that the systems PTS and PTS_f are equivalent. We finish by proving an injectivity statement for products.

In chapter 6 we will describe the formalisation of all results in chapter 5. Lemmas and Theorems which have been formalised will state the name of the result in the Coq code using the format `[Coq name]`.

In chapter 7 we look more closely at the particular PTS $\lambda_f \mathbf{P}$ (which is our notation of λF used in [Geuvers and Wiedijk, 2008]) 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 used rules for the PTS in these particular cases.

2 PTS: 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 thesis 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.

2.1 Pseudoterms

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 $\mathbf{S}$, 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*:

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

This means that

- if $x \in \mathcal{V}$ then $x \in \mathcal{T}$ and similarly if $s \in \mathcal{S}$ then $s \in \mathcal{T}$;
- if $a_1, a_2 \in \mathcal{T}$, the *application* $a_1 a_2 \in \mathcal{T}$. One should think of an application as the image of the value a_1 under the function a_2. We will usually abbreviate $(a_1 a_2)a_3$ to $a_1 a_2 a_3$;
- if $a_1, a_2 \in \mathcal{T}$ and $x \in \mathcal{V}$, the *product* $\Pi x{:}a_1.a_2$ and the *abstraction* $\lambda x{:}a_1.a_2$ are elements of $\mathcal{T}$. In both cases all occurrences of x in a_2 are *bound* by this product/abstraction (unless they were already bound in another product or abstraction). All occurrences of variables that are not bound are called *free*. The set of bound variables and free variables occurring in a is denoted by $\mathrm{BV}(a)$ resp. $\mathrm{FV}(a)$. One should think of $\lambda x{:}A.b$ as the function $x \mapsto b$ (note that the variable x can occur in b) with domain A and $\Pi x{:}A.B$ as the set-theoretical product $\prod_{x \in A} B$. If B does not depend on x, then we simplify the notation $\Pi x{:}A.B$ to $A \to B$, which is the function type from A to B. We abbreviate $A \to (B \to C)$ to $A \to B \to C$.
- $\mathcal{T}$ is the smallest such set.

We will use the letters $a, b, c, d, A, B, C, D, M, N$, possibly adorned with primes or subscripts, for pseudoterms. An important convention is that we identify terms up to *alpha convertibility.* To define this we first define the *substitution* $a_1[x := a_2]$ (with a_1 and a_2 pseudoterms and x a variable) to mean the pseudoterm a_1, where we replace each free occurrence of x by a_2. This is only defined if no free variable in a_2 becomes bound in $a_1[x := a_2]$. Alpha convertibility is the transitive compatible closure of the following relation (for any pseudoterms where the substitution is defined)

$$\begin{aligned}\lambda x{:}a_1.a_2 &\equiv \lambda y{:}a_1.(a_2[x := y]);\\ \Pi x{:}a_1.a_2 &\equiv \Pi y{:}a_1.(a_2[x := y]).\end{aligned} \tag{2}$$

The *compatible closure* of a relation $\sim$ is the smallest relation respecting the structure of pseudoterms. This means that if $M \sim M'$ then the following six relations also hold:

$$\begin{array}{lll} MN \sim M'N, & \Pi x{:}M.N \sim \Pi x{:}M'.N, & \lambda x{:}M.N \sim \lambda x{:}M'.N \\ NM \sim NM', & \Pi x{:}N.M \sim \Pi x{:}N.M', & \lambda x{:}N.M \sim \lambda x{:}N.M'. \end{array}$$

So alpha convertibility is the smallest transitive relation respecting the structure of pseudoterms containing all relations of the form (2), and we write alpha convertibility by $\equiv$. One can show that alpha convertibility is in fact an equivalence relation, and we identify two terms which are alpha convertible. So in fact we look at equivalence classes of pseudoterms. This has the advantage that we can define the substitution $a_1[x := a_2]$ for any two pseudoterms a_1 and a_2. Because if the substitution $a_1[x := a_2]$ is invalid, meaning that some free variable in a_2 become bound in $a_1[x := a_2]$, we can replace a_1 with an alpha-equivalent a'_1 by replacing all bound variables with fresh variables. Then one can ensure that $\mathrm{FV}(a_2) \cap \mathrm{BV}(a'_1) = \varnothing$, which implies that the substitution is well-defined. Throughout the rest of this thesis, we'll write a pseudoterm for the corresponding equivalence class, and do not make a distinction between a_1 and a'_1. This does not lead to complications, because all functions and relations we define on pseudoterms, respect alpha convertibility. If we remember our intuition that abstractions are functions, this identification of terms with equivalence classes does make sense. In ordinary mathematics, we also do not distinguish between the functions $x \mapsto f(x)$ and $y \mapsto f(y)$ or between the product sets $\Pi_{x \in A} B(x)$ and $\Pi_{y \in A} B(y)$.

The next relation we define is beta reduction. *One step beta reduction* is the compatible closure of the relation

$$(\lambda x{:}A.M)N \rightsquigarrow_\beta M[x := N] \tag{3}$$

and denoted by $\to_\beta$. *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.

Theorem 2.1.

1. *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. *If for pseudoterms A, B we have $A \simeq_\beta B$ then there is a pseudoterm C such that $A \twoheadrightarrow_\beta C$ and $B \twoheadrightarrow_\beta C$.*

Proof. It is easy to prove equivalence of the statements, but proving either statement is not so easy. We will skip the proof here, but in [Takahashi, 1995] a proof is given using the technique of parallel beta reduction. □

Example 2.2. If x, y, z, f, A, B, P are variables and $*$ is a sort, examples of pseudoterms are

$$x; \qquad (\lambda x{:}A.y)z; \qquad \lambda A{:}*.\lambda f{:}A \to A.\lambda x{:}A.f(fx); \qquad \lambda P{:}*.\Pi x{:}A.\Pi y{:}B.Pxy.$$

Note that $A \to A$ is short for $\Pi z{:}A.A$ and that Pxy is short for $(Px)y$. These pseudoterms are actually all terms (for a suitable specification), which means that they are all well-behaved. We will explain later what we exactly mean by that. An example of a pseudoterm which is not a term is $\lambda f{:}A.\lambda x{:}B.fxx$, because here the we apply the pseudoterm f of type A to x twice assuming that f is a function which accept terms from type B. But the type of f does not give any hint that this is the case, and indeed this pseudoterm is not a term.

As an example of alpha conversion, note that $\lambda P{:}*.\Pi x{:}A.Px \equiv \lambda Q{:}*.\Pi y{:}A.Qy$. For beta conversion, we have $(\lambda x{:}A.y)z \to_\beta y$. A more complicated example is

$$(\lambda x{:}A.xx)(\lambda y{:}B.y) \to_\beta (\lambda y{:}B.y)(\lambda y{:}B.y) \to_\beta \lambda y{:}B.y.$$

This means that $(\lambda x{:}A.xx)(\lambda y{:}B.y) \twoheadrightarrow_\beta \lambda y{:}B.y$. Because also $(\lambda z : C.\lambda y{:}B.y)a \to_\beta \lambda y{:}B.y$, we conclude that $(\lambda x{:}A.xx)(\lambda y{:}B.y) \simeq_\beta (\lambda z : C.\lambda y{:}B.y)a$. ∅

2.2 Judgements

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

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

$$\frac{}{\cdot \vdash} \tag{nil}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash} \ x \notin \operatorname{dom}\Gamma \tag{cons}$$

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

$$\frac{\Gamma \vdash}{\Gamma \vdash x : A} \ (x : A) \in \Gamma \tag{var}$$

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

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

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

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

Figure 2.1: rules for a PTS $\lambda\mathbf{S}$

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

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

for some $n \geq 0$. We will not write the dot in this notation: $\Gamma \equiv x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$. We define $\operatorname{dom}\Gamma = \{x_1, \ldots, x_n\}$ and $(x : A) \in \Gamma$ if $x = x_i$ and $A \equiv A_i$ for some i. We write $\Gamma[x := a]$ for the context $x_1 : A_1[x := a], x_2 : A_2[x := a], \ldots, x_n : A_n[x := a]$ (we will only use this when $x \neq x_i$ for all i). Given two contexts Γ and Δ, we will write the concatenation of these contexts simply by Γ, Δ. One should think of a context as a sequence of variable declarations, i.e. specifying that the variable x_1 has type A_1, variable x_2 has type A_2, and so on.

Furthermore we have *pseudojudgements* of the form

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

One should think of the *legality* pseudojudgement $\Gamma \vdash$ as "Γ is a legal context" and the *typing* pseudojudgement $\Gamma \vdash a : A$ as the statement "term a has type A in context Γ" or more loosely "term a is an element of set A when the variables in dom(Γ) live inside the sets specified by Γ." As explained in the introduction, a judgement can also mean "a is a proof of proposition A." A judgement with a sort as type, like $\Gamma \vdash B : s$, can be interpreted as "B is a valid type under context Γ (in universe s)."

Example 2.3. If $*$ is a sort and $\mathbb{N}, 0, S$ are variables, then $\Gamma \equiv \mathbb{N} : *, 0 : \mathbb{N}, S : \mathbb{N} \to \mathbb{N}$ is a pseudocontext. Now $\Gamma \vdash$ and $\Gamma \vdash S(S0) : \mathbb{N}$ are pseudojudgements. $\varnothing$

We use the abbreviation $\Gamma \vdash A : B : C$ for "$\Gamma \vdash A : B$ and $\Gamma \vdash B : C$."

Now we define the set of judgements recursively. The rules generating the judgements are given in Figure 2.1. This means that the set of judgements is the smallest set which is closed under all the rules in Figure 2.1. All rules are of the form

$$\frac{\text{zero or more judgements as hypothesis}}{\text{conclusion}} \ \text{(possible other hypothesis).}$$

We will call the pseudocontexts and pseudoterms occurring in these judgements contexts resp. terms. The letters s or s_i refer to sorts, the letter x to variables and all other letters can be any term. We will now treat all rules briefly.

- The rule (nil) is the only rule without another judgement as hypothesis, so it is the top of every branch in a derivation of a judgement. It just states that the empty context is legal.
- The rule (cons) describes when the nonempty contexts are legal. It states that to add $(x : A)$ to Γ we need to ensure that A can be typed by a sort, and that x does not already occur in Γ.

  All other rules are typing rules. Except for (conv) all rules type a unique kind of term (for example (var) types variables and (app) types applications).
- The first typing rule is (sort) which states how we get judgements from the axioms $\mathcal{A}$.
- (var) types variables. A variable has a type if it occurs in the context and if the context is legal
- The rule (prod) states how to type products. For this the relations $\mathcal{R}$ of the specification are important, they determine what type A and B should have for the product $\Pi x{:}A.B$ to be well typed.
- The rule (abs) describes how abstractions are typed. The type of an abstraction is a product. Note that the hypotheses of the abs-rule are also sufficient to type the product $\Pi x{:}A.B$.
- The (app)-rule specifies how to type applications. To understand why Fa has this type, remember that we viewed F as a function sending a in A to Fx in $B(a)$, and $B(a)$ is written as $B[x := a]$ using substitution. The analogue in set theory is that if $F \in \Pi_{x\in A}B(x)$, then $F(a) \in B(a)$.
- Finally we have the conversion rule (conv). This states loosely that whether two types are beta convertible, then they contain the same terms in them.

Example 2.4. The specification $\mathbf{P}$ is defined in the following way. $\mathbf{P} = (\mathcal{S}_\mathbf{P}, \mathcal{A}_\mathbf{P}, \mathcal{R}_\mathbf{P})$ where

$$\begin{aligned}\mathcal{S}_\mathbf{P} &= \{*, \square\};\\ \mathcal{A}_\mathbf{P} &= \{(*, \square)\};\\ \mathcal{R}_\mathbf{P} &= \{(*,*,*), (*,\square,\square)\}.\end{aligned}$$

This is the specification of $\lambda\mathbf{P}$ in the lambda cube (Figure 1.1). The other systems of the lambda cube have the same sorts and axioms, but different relations.

- All systems of the lambda cube have the relation $(*,*,*)$.
- The systems on the right side of the cube (λP, $\lambda P\underline{\omega}$, $\lambda P2$ and λC) have the relation $(*,\square,\square)$.
- The systems on the back of the cube ($\lambda\underline{\omega}$, $\lambda P\underline{\omega}$, $\lambda\omega$ and λC) have the relation $(\square,\square,\square)$.
- The systems on the top of the cube ($\lambda 2$, $\lambda P2$, $\lambda\omega$ and λC) have the relation $(\square,*,*)$.

The following are judgements in $\lambda\mathbf{P}$.

$$\begin{array}{cc} \cdot \vdash * : \square & A : * \vdash \lambda x{:}A.x : A \to A \\ A : * \vdash A : * & A : *, F : A \to *, f : \Pi x{:}A.Fx \vdash \\ A : *, x : A \vdash & A : *, F : A \to *, f : \Pi x{:}A.Fx \vdash \lambda x{:}A.fx : \Pi x{:}A.Fx : * \\ A : * \vdash A \to * : \square & A : *, a : A, b : A \vdash (\lambda x{:}A.a)b : A \end{array}$$

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

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

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

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

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

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

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

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

Also note that the following judgement holds

$$A : *, a : A, F : A \to *, g : A \to A, f : (\lambda x{:}A.F(g\,x))a \vdash f : (\lambda x{:}A.F(g\,x))a,$$

hence by the conversion rule we also have

$$A : *, a : A, F : A \to *, g : A \to A, f : (\lambda x{:}A.F(g\,x))a \vdash f : F(g\,a).$$

One can now also verify that the judgements written in the introduction are valid in $\lambda\mathbf{P}$. $\varnothing$

2.3 Alternative rules

The rules we gave for judgements are 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, there's no judgement dedicated to stating that a context is legal. Note that the rules (prod), (abs), (app) and (conv) are exactly the same. In this set of rules, the (ax)-rule is used to type sorts, but this is only done for the empty context. The (var′)-rule types variables and also adds an term to the context. The (weak)-rule is used to add a term to the context while preserving the rest of the judgement.

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

Proposition 2.5. *The rules in Figure 2.1 define the same typing judgements as the rules in Figure 2.2.*

Before we give a proof, we need some basic properties about each set of rules. To distinguish between the sets of rules we write a judgement derived from the rules in Figure 2.2 with a prime in this section, i.e. $\Gamma \vdash' M : A$. Now we have to prove that $\Gamma \vdash M : A$ iff $\Gamma \vdash' M : A$.

We will use *structural induction* on the derivation of the judgement. This means that given a proposition P on judgements, we prove for every rule that if P holds for all hypotheses, then it also holds for the conclusion. This implies that the statement holds for all judgements.

We need the following Lemmas before we can give the proof.

Lemma 2.6. *If $\Gamma \vdash M : A$ then $\Gamma \vdash$.*

Proof. We will later prove the same Lemma for a variant of PTS in Lemma 5.3. The proof is exactly the same, so we will skip it here. □

Lemma 2.7 (Weakening)**.** *If $\Gamma, \Delta \vdash M : A$ and $\Gamma \vdash_f B : s$ and $x \notin \mathrm{dom}(\Gamma, \Delta)$ then $\Gamma, x : B, \Delta \vdash M : A$.*

Proof. We will also prove this lemma later for a variant, which is Lemma 5.4. □

Lemma 2.8 (Start Lemma for the Alternative Rules)**.** *If $\Gamma \vdash' M : A$ then $\Gamma \vdash' s : t$ for all axioms $(s, t) \in \mathcal{A}$ and $\Gamma \vdash' x : B$ for all $(x : B) \in \Gamma$.*

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

If the last rule was (ax), then $\Gamma \equiv \cdot$, and both conclusions are trivial.

If the last rule was either (var) or (weak), then $\Gamma \equiv \Gamma', x' : A'$ for some A', and one of the hypotheses of the rule was $\Gamma' \vdash' A' : s'$ for some sort s'. Now if $(s, t) \in \mathcal{A}$, then by induction hypothesis we know that $\Gamma' \vdash' s : t$, so by (weak) we conclude that $\Gamma \vdash' s : t$, proving the first part. Now suppose $(x : B) \in \Gamma$, then either $B \equiv A'$ and $x' \equiv x$ or $(x : B) \in \Gamma'$. In the first case we can conclude $\Gamma \vdash' x : B$ by (var), and in the second case we know by the IH (induction hypothesis) that $\Gamma' \vdash' x : B$, and we can conclude that $\Gamma \vdash' x : B$ by (weak).

If the last rule was any other rule, then it contains some assumption with context Γ, so the statement follows from the IH. □

Now we can give the proof that the sets of rules are equivalent.

Proof (of Proposition 2.5). We first prove that $\Gamma \vdash' M : A$ implies $\Gamma \vdash M : A$ by induction on the derivation of the hypothesis.

If the last rule was (ax), then we can first apply (nil) and then (sort) to get the desired judgement.

If the last rule was (var), then we know that $\Gamma \equiv \Gamma', x : A$ and that $\Gamma' \vdash' A : s$ is an hypothesis. By the IH we know that $\Gamma' \vdash A : s$, hence by Lemma 2.6 also that $\Gamma' \vdash$. Hence by (cons) we know that $\Gamma \vdash$ so by (var) that $\Gamma \vdash x : A$.

If the last rule was (weak), one can apply Lemma 2.7 to the IH of the second hypothesis to obtain the desired result.

All other rules are exactly the same for both systems, hence the result follows for these rules by applying the same rule to the IH.

Next we prove $\Gamma \vdash M : A$ implies $\Gamma \vdash' M : A$. But we cannot do this simply by induction on the typing judgements, because we also need to know what the equivalent statement to $\Gamma \vdash$ is in the alternative set of rules. One might be tempted that it is equivalent to $\exists M, A\colon \Gamma \vdash' M : A$, which is almost correct. There's one subtlety, which is that the set of axioms $\mathcal{A}$ might be empty. Then neither system can prove any typing judgements, but we can still prove $\cdot \vdash$. So the correct implication is the following:

- If $\Gamma \vdash M : A$ then $\Gamma \vdash' M : A$;
- If $\Gamma \vdash$ then either $\Gamma \equiv \cdot$ or there exists pseudoterms M, A such that $\Gamma \vdash' M : A$.

We prove the combination of these statements by induction over the derivation of the hypothesis.

If the last rule was (nil), then $\Gamma \equiv \cdot$ and we are done.

If the last rule was (cons), then it was concluded from $\Gamma' \vdash A : s$ for some Γ', A, s. By the IH we conclude that $\Gamma' \vdash' A : s$, and by (var) we conclude that $\Gamma \vdash x : A$ and we are done.

If the last rule was (sort), then by the IH on the hypothesis we conclude that either $\Gamma \equiv \cdot$ or $\Gamma \vdash' M : A$ for some M, A. In the first case, we can just apply (ax), and in the second case, we can apply Lemma 2.8 to conclude the desired result.

If the last rule was (var), then we can apply IH on the hypothesis again to conclude either $\Gamma \equiv \cdot$ or $\Gamma \vdash' M : A$. The first case is impossible, since $(x : A) \in \Gamma$, and the second case follows by applying Lemma 2.8.

This finishes the prove of $\Gamma \vdash' M : A$ iff $\Gamma \vdash M : A$. □

For a more detailed overview of PTSs, see [Barendregt, 1992]. In this book, the rules in Figure 2.2 are used. Then the meta-theory of these rules is developed. In this thesis we will use one nontrivial proposition from this meta-theory.

Theorem 2.9 (Subject Reduction)**.** *If* $\Gamma \vdash A : B$ *and* $A \twoheadrightarrow_\beta A'$ *then* $\Gamma \vdash A' : B$.

Proof. See [Barendregt, 1992], Lemma 5.2.15 on page 107. □

3 PTS$_e$: Typed judgemental equality

Given a specification $\mathbf{S}$, 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 ($\vdash$) 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 (3) 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 [Adams, 2006] it is shown that these two different type systems are equivalent for so-called functional specifications (cf. Definition 7.2). In [Siles and Herbelin, 2012] this equivalence is generalised to arbitrary specifications. The equivalence is formulated as follows.

Theorem 3.1 (Equivalence of PTS and PTS$_e$)**.**

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 [Siles and Herbelin, 2012].

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 [Adams, 2006]. One could try to derive that if $\Gamma \vdash_e M : A$ 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 the usual 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 $x_1 : A_1, \dots, x_n : A_n \to_\beta x_1 : B_1, \dots, x_n : B_n$ 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. In fact, it turns out that this version of product injectivity is too strong, and does not hold in general [Siles and Herbelin, 2012].

The way Siles and Herbelin 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 [Takahashi, 1995]. 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$_e$. This equivalence implies Theorem 3.1.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.1: rules for a PTS$_e$ $\lambda_e\mathbf{S}$

4 PTS$_f$: Typed convertibility proofs

For a specification **S** we define the *Pure Type System with convertibility proofs* $\lambda_f\mathbf{S}$ as follows. There is a separate class $\mathcal{H}$ of pseudoconvertibility 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):[5]

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

We define $H[x := a]$ in the obvious way, by replacing x with a for every free occurrence of x in H. 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 $[x : A]$.

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* [McBride, 2002]. In fact, we will show an example of an equality between terms which do not have the same type in Section 7.1.

In summary, the judgements have the grammar

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

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 the term exactly tells which rules were used to derive the equality. Most of the rules for equality judgements correspond to a similar PTS$_e$-rule. There are again rules for reflexivity, symmetry and transitivity. Then we have the beta rule, and rules to equate products, abstractions and applications. Note that we need many more hypotheses to these rules than for the rules for PTS$_e$, because we need both typing information and equality information as hypotheses. In the rules for PTS$_e$, these could be given in a single judgement, unlike PTS$_f$. At last we have the (iota)-rule which describes the equality between a term and the same term annotated with a convertibility proof. In Chapter 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 PTS$_f$ is the following Theorem. The name written in format `[Coq name]` refers to the name in the formalisation.

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* $\mathrm{der}(J)$ (which a priori depends on more than only J) with nodes labelled by $\{(\mathrm{nil}), (\mathrm{cons}), (\mathrm{sort}), \ldots, (\mathrm{iota})\}$, describing which rules are used. For example $\mathrm{der}(\cdot \vdash_f)$ is a tree with the single node labelled 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

$$\mathrm{der}(\Gamma \vdash_f \lambda x{:}A.b : \Pi x{:}A.B) = \begin{tikzcd}[column sep=small] \mathrm{der}(\Gamma \vdash_f A : s_1) \arrow[dr, no head] & \mathrm{der}(\Gamma, x : A \vdash_f b : B) \arrow[d, no head] & \mathrm{der}(\Gamma, x : A \vdash_f B : s_2) \arrow[dl, no head] \\ & (\mathrm{abs}) & \end{tikzcd}$$

[5]The ι in the grammar has nothing to do with ι-reduction.

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{ll} \Gamma \vdash_f A : s_1 & \Gamma, x : A \vdash_f B : s_2 \\ \Gamma \vdash_f A' : s_1' & \Gamma, x' : A' \vdash_f B' : s_2' \\ \Gamma \vdash_f H : A = A' & \Gamma, x : A \vdash_f H' : B = B'[x' := x^H] \end{array}}{\Gamma \vdash_f \{H, [x : A]H'\} : \Pi x{:}A.B = \Pi x'{:}A'.B'}\; \begin{array}{l} (s_1, s_2, s_3) \in \mathcal{R} \\ (s_1', s_2', s_3') \in \mathcal{R} \end{array} \qquad \text{(prod-eq)}$$

$$\frac{\begin{array}{ll} \Gamma \vdash_f A : s_1 & \Gamma, x : A \vdash_f b : B : s_2 \\ \Gamma \vdash_f A' : s_1' & \Gamma, x' : A' \vdash_f b' : B' : s_2' \\ \Gamma \vdash_f H : A = A' & \Gamma, x : A \vdash_f H' : b = b'[x' := x^H] \end{array}}{\Gamma \vdash_f \langle H, [x : A]H' \rangle : \lambda x{:}A.b = \lambda x'{:}A'.b'}\; \begin{array}{l} (s_1, s_2, s_3) \in \mathcal{R} \\ (s_1', s_2', s_3') \in \mathcal{R} \end{array} \qquad \text{(abs-eq)}$$

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

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

Figure 4.1: rules for a PTS$_f$ $\lambda_f\mathbf{S}$

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

We won't prove this Theorem yet, because we first need to develop some of the meta-theory. It will be proven in Section 5.3.

Definition 4.3. We define the following concepts for a PTS$_f$ $\lambda_f\mathbf{S}$:

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 in which M appears 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:

$$\begin{array}{lll} |s| \equiv s & |\Pi x{:}A.B| \equiv \Pi x{:}|A|.|B| & |Fa| \equiv |F||a| \\ |x| \equiv x & |\lambda x{:}A.b| \equiv \lambda x{:}|A|.|b| & |a^H| \equiv |a| \end{array}$$

   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 is 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

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

To avoid making lemmas unnecessarily long we will use the following convention: if a symbol is not introduced in the statement, it is considered to be universally quantified at the start of the lemma. For example with this convention, Theorem 2.1.1 could have been formulated as

If $A \twoheadrightarrow_\beta B$ and $A \twoheadrightarrow_\beta C$, then there is a D such that $B \twoheadrightarrow_\beta D$ and $C \twoheadrightarrow_\beta D$.

5 Meta-theory of PTS$_f$

In this chapter we will 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 prove some properties about variables occurring in judgements.

In Section 5.2 we will prove some standard Lemmas of the meta-theory of $\lambda_f\mathbf{S}$ which also have counterparts in $\lambda\mathbf{S}$ and $\lambda_e\mathbf{S}$.

In Section 5.3 we will prove that every judgement has a unique derivation.

In Section 5.4 we will prove the implication $\lambda_f\mathbf{S} \Rightarrow \lambda\mathbf{S}$ and some properties about the erasure map.

In Section 5.5 we will prove a Lemma about equality between substitutions.

Finally in Section 5.6 we prove the implication $\lambda_e\mathbf{S} \to \lambda_f\mathbf{S}$, and Product Injectivity as Corollary.

All results in Sections 5.2-5.6 have been formalised in Coq. The Lemmas in 5.1 have not been formalised, since we used a different system for writing variables in the formalisation (see Chapter 6).

5.1 Properties of variables

Lemma 5.1 (Free Variables). *Let $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$ be a context which appears in a judgement. Then*

1. *the variables $x_1, \ldots, x_n$ are distinct;*
2. *for all $k \leq n$ we have $\mathrm{FV}(A_k) \subseteq \{x_1, \ldots, x_{k-1}\}$;*
3. *if $\Gamma \vdash_f M : A$ then $\mathrm{FV}(M), \mathrm{FV}(A) \subseteq \mathrm{dom}\,\Gamma$;*
4. *if $\Gamma \vdash_f H : M = N$ then $\mathrm{FV}(H), \mathrm{FV}(M), \mathrm{FV}(N) \subseteq \mathrm{dom}\,\Gamma$.*

Proof. We prove all statements simultaneously by induction on the derivation of the judgement. We distinguish cases according to the last applied rule. We write IHn ($n \in \{1,2,3,4\}$) for the induction hypothesis of statement n. Note that not all statements are meaningful for every induction step. For example if we derived $\Gamma \vdash_f$ from (nil) or (cons), only 1 and 2 need a proof.

If the last rule was (nil), then 1 and 2 are vacuously true since the context is empty.

If the last rule was (cons), then the rule enforces that x_n is is not in $\{x_i \mid i < n\}$ and by IH1 the x_i are distinct for $i < n$. Statement 2 follows from IH2 and IH3.

For all other rules, the context in the conclusion of the rule also appears in one of the assumptions. For this reason statements 1 and 2 follow directly from IH1 and IH2. Statements 3 and 4 also follow easily; we proof just (var) and (app) here.

If the last rules was (var), concluding $\Gamma \vdash_f x : A$ from $\Gamma \vdash_f$ and $(x : A) \in \Gamma$, then $\mathrm{FV}(x) = \{x\} \subseteq \mathrm{dom}\,\Gamma$ by assumption of the rule and $\mathrm{FV}(A) \subseteq \mathrm{dom}\,\Gamma$ follows from IH2.

If the last rule was (app), concluding $\Gamma \vdash_f Fa : B[x := a]$ from $\Gamma \vdash_f F : \Pi x{:}A'.B$ and $\Gamma \vdash_f a : A'$, then by IH3 we know that $\mathrm{FV}(F) \subseteq \mathrm{dom}\,\Gamma$, $\mathrm{FV}(a) \subseteq \mathrm{dom}\,\Gamma$ and

$$\mathrm{FV}(B) \setminus \{x\} \subseteq \mathrm{FV}(A') \cup (\mathrm{FV}(B) \setminus \{x\}) = \mathrm{FV}(\Pi x{:}A'.B) \subseteq \Gamma.$$

Now the statement follows by noting that $\mathrm{FV}(B[x := a]) \subseteq (\mathrm{FV}(B) \setminus \{x\}) \cup \mathrm{FV}(a)$ and $\mathrm{FV}(Fa) = \mathrm{FV}(F) \cup \mathrm{FV}(a)$. □

We have alpha-equivalence for terms, but not for judgements. So the judgements $y : s \vdash_f y : s$ and $x : s \vdash_f x : s$ are considered different judgements. We can use the following Lemma to show that these judgements are equivalent. We need this Lemma later to ensure that some variables occurring in different judgements are different.

Lemma 5.2 (Alpha Conversion for Judgements).

1. *If $\Gamma, x : T, \Delta \vdash_f$ and $y \notin \mathrm{dom}(\Gamma, x : T, \Delta)$, then $\Gamma, y : T, \Delta[x := y] \vdash_f$;*

2. *If* $\Gamma, x : T, \Delta \vdash_f M : N$ *and* $y \notin \mathrm{dom}(\Gamma, x : T, \Delta)$*, then* $\Gamma, y : T, \Delta[x := y] \vdash_f M[x := y] : N[x := y]$*;*

3. *If* $\Gamma, x : T, \Delta \vdash H : M = N$ *and* $y \notin \mathrm{dom}(\Gamma, x : T, \Delta)$*, then* $\Gamma, y : T, \Delta[x := y] \vdash_f H[x := y] : M[x := y] = N[x := y]$*.*

Proof. The proof is easy and not important, so we skip it. □

5.2 Basic properties of PTS$_f$

Lemma 5.3. `[wf_typ]` *If* $\Gamma \vdash_f M : N$ *then* Γ *is legal.*

Proof. We will use induction on the derivation of the judgement $\Gamma \vdash_f M : N$, distinguishing cases according to the last applied rule.

If the last rule was (sort) or (var), then it is an assumption of the rule. If the last rule was any other rule, then Γ also appears in a typing judgement as assumption of the rule, hence the statement follows by induction hypothesis. □

Lemma 5.4 (Weakening)**.** `[weakening]`

1. *If* $\Gamma, \Delta \vdash_f$*,* $\Gamma \vdash_f T : s$ *and* $x \notin \mathrm{dom}(\Gamma, \Delta)$ *then* $\Gamma, x : T, \Delta \vdash_f$*;*

2. *If* $\Gamma, \Delta \vdash_f M : N$*,* $\Gamma \vdash_f T : s$ *and* $x \notin \mathrm{dom}(\Gamma, \Delta)$ *then* $\Gamma, x : T, \Delta \vdash_f M : N$*;*

3. *If* $\Gamma, \Delta \vdash_f H : M = N$*,* $\Gamma \vdash_f T : s$ *and* $x \notin \mathrm{dom}(\Gamma, \Delta)$ *then* $\Gamma, x : T, \Delta \vdash_f H : M = N$*.*

Proof. We will prove these statements by simultaneous induction on the derivation of the first judgement in each item, distinguishing cases according to the last applied rule.

If the last rule was (nil), then by assumption $\cdot \vdash_f T : s$, so by (cons), the context $x : T$ is legal.

If the last rule was (cons), then we distinguish two cases. If $\Delta = \cdot$, then by (cons) we know that $\Gamma, x : T$ is legal. Otherwise, say $\Delta = \Delta', y : A$. In this case we can apply the IH to the assumption $\Gamma, \Delta' \vdash_f A : s$ of (cons), obtaining $\Gamma, x : T, \Delta' \vdash_f A : s$. We finish by applying (cons).

In all other cases you can use the induction hypothesis on all assumptions and then use the same rule. The only complication which can arise is that you have to rename variables in contexts which do not occur in the context of the conclusion. We'll treat (prod) to illustrate this.

Suppose we concluded $\Gamma, \Delta \vdash_f \Pi y{:}A.B : s_3$ from $\Gamma, \Delta \vdash_f A : s_1$ and $\Gamma, \Delta, y : A \vdash_f B : s_2$. We can replace all occurrences of y with a fresh variable z by Lemma 5.2. Now one can apply the IH to each assumption, obtaining $\Gamma, x : T, \Delta \vdash_f A : s_1$ and $\Gamma, x : T, \Delta, z : A \vdash_f B[y := z] : s_2$ and by (prod) we conclude $\Gamma, x : T, \Delta \vdash_f \Pi z{:}A.B[y := z] : s_3$ hence also $\Gamma, x : T, \Delta \vdash_f \Pi y{:}A.B : s_3$ by alpha-equivalence. □

Lemma 5.5 (Thinning)**.** `[thinning_n]` *If* $\Gamma \vdash_f M : N$ *and* Γ, Δ *is legal, then* $\Gamma, \Delta \vdash_f M : N$*.*

Proof. We will prove this by induction on the length of Δ.

If $\Delta = \cdot$, then the statement is trivial.

If $\Delta = \Delta', x : A$, then the statement that Γ, Δ is legal can only be concluded from (cons), with assumption $\Gamma, \Delta' \vdash_f A : s$ for some sort s. By Lemma 5.3 we know that Γ, Δ' is legal, hence by the IH we know that $\Gamma, \Delta' \vdash_f M : N$. By applying Weakening (Lemma 5.4) we derive that $\Gamma, \Delta \vdash_f M : N$, as desired. □

Lemma 5.6 (Substitution)**.** `[substitution]`

1. *If* $\Gamma, x : A, \Delta \vdash_f$ *and* $\Gamma \vdash_f a : A$ *then* $\Gamma, \Delta[x := a] \vdash_f$*.*

2. *If* $\Gamma, x : A, \Delta \vdash_f B : C$ *and* $\Gamma \vdash_f a : A$ *then* $\Gamma, \Delta[x := a] \vdash_f B[x := a] : C[x := a]$*.*

3. *If* $\Gamma, x : A, \Delta \vdash_f H : B = C$ *and* $\Gamma \vdash_f a : A$ *then* $\Gamma, \Delta[x := a] \vdash_f H[x := a] : B[x := a] = C[x := a]$*.*

Proof. We will prove these statements by simultaneous induction on the derivation of the first judgement in each item, distinguishing cases according to the last applied rule.

The last rule cannot be (nil).

If the last rule was (cons), then we distinguish the cases $\Delta = \cdot$ and $\Delta = \Delta', y : D$. In the first case, the assumption to the rule is $\Gamma \vdash_f A : s$ and we have to prove that Γ is legal, which is true by Lemma 5.3. In the second case, the assumption was $\Gamma, x : A, \Delta' \vdash_f D : s$ and by the IH we know that $\Gamma, \Delta'[x := a] \vdash_f D[x := a] : s$, hence by (cons) we conclude that $\Gamma, \Delta'[x := a], y : D[x := a] \equiv \Gamma, \Delta[x := a]$ is legal, as desired.

If the last rule is (sort), we conclude by the IH that $\Gamma, \Delta[x := a]$ is legal, hence by applying (sort) we finish the step.

If the last rule is (var), concluding $\Gamma, x : A, \Delta \vdash_f y : C$ then we also conclude that $\Gamma, \Delta[x := a]$ is legal. We distinguish three cases, and we use several parts of Lemma 5.1.

- If $(y : C) \in \Gamma$, then we conclude $\Gamma, \Delta[x := a] \vdash_f y : C$, as desired.
- If $y = x$, then $C \equiv A$, and we know that $\Gamma \vdash_f a : A$, hence by Thinning (Lemma 5.5) we conclude $\Gamma, \Delta[x := a] \vdash_f a : A$, as desired.
- If $(y : C) \in \Delta$, then we know that $(y : C[x := a]) \in \Delta[x := a]$, hence by (var) we conclude $\Gamma, \Delta[x := a] \vdash_f y : C[x := a]$, as desired.

Most other induction steps are similar to each other. For this reason, we only treat 3 of them.

If the last rule was (conv), then the judgement is of the form $\Gamma, x : A, \Delta \vdash_f b^H : C$ concluded from $\Gamma, x : A, \Delta \vdash_f b : B$ and $\Gamma, x : A, \Delta \vdash_f H : B = C$. By induction hypothesis we have $\Gamma, \Delta[x := a] \vdash_f b[x := a] : B[x := a]$ and $\Gamma, \Delta[x := a] \vdash_f H[x := a] : B[x := a] = C[x := a]$, so the conclusion follows by applying (conv).

If the last rule was (beta), then $\Gamma, x : A, \Delta \vdash_f \beta((\lambda y{:}B.c)b) : (\lambda y{:}B.c)b = c[y := b]$ was concluded from $\Gamma, x : A, \Delta \vdash_f b : B : s_1$ and $\Gamma, x : A, \Delta, y : B \vdash_f c : C : s_2$ for some relation (s_1, s_2, s_3). By the IH we know that $\Gamma, \Delta[x := a] \vdash_f b[x := a] : B[x := a] : s_1$ and $\Gamma, \Delta[x := a], y : B[x := a] \vdash_f c[x := a] : C[x := a] : s_2$, so by the (beta) rule we conclude

$$\begin{aligned}\Gamma, \Delta[x := a] \vdash_f \beta((\lambda y{:}B[x := a].c[x := a])b[x := a]) :& \\ (\lambda y{:}B[x := a].c[x := a])b[x := a] &= c[x := a][y := b[x := a]].\end{aligned}$$

Note that $c[x := a][y := b[x := a]] = c[y := b][x := a]$, so we conclude

$$\Gamma, \Delta[x := a] \vdash_f \beta((\lambda y{:}B.c)b)[x := a] : ((\lambda y{:}B.c)b)[x := a] = c[y := b][x := a],$$

as desired.

If the last rule was (abs-eq), then the judgment $\Gamma, x : A, \Delta \vdash_f \langle H, [y : B]H' \rangle : \lambda y{:}B.c = \lambda y'{:}B'.c'$ was concluded from six assumptions. Using the IH on all these assumptions and noting that $b'[y' := y^H][x := a] = b'[x := a][y' := y^{H[x:=a]}]$ we can use the rule (abs-eq) again to conclude

$$\Gamma, \Delta \vdash_f \langle H[x := a], [y : B[x := a]]H'[x := a] \rangle : \lambda y{:}B[x := a].c[x := a] = \lambda y'{:}B'[x := a].c'[x := a],$$

which is the conclusion we wanted. □

Usually in the basic meta-theory of a PTS there is a Generation Lemma stating how one could have derived the typing of a certain term. In this variant of PTS such a Lemma is not necessary. If one has a judgement, there is a unique last rule which could have been used to derive the judgement. So a Generation Lemma does not add much.

We do need the following Lemma which is a bit like a Generation Lemma for contexts.

Lemma 5.7 (Context Generation). `[wf_item]` *If* $\Gamma, x : A, \Delta$ *is legal, then* $\Gamma \vdash_f A : s$ *for some sort* s*.*

Proof. We use simultaneous induction on the derivation of the assumptions in the following two statements

- If $\Gamma, x : A, \Delta \vdash_f M : N$, then $\Gamma \vdash_f A : s$ for some sort s.
- If $\Gamma, x : A, \Delta \vdash_f$, then $\Gamma \vdash_f A : s$ for some sort s.

The last rule cannot be (nil).

If the last used rule is (cons), then we distinguish the cases $\Delta = \cdot$ and $\Delta = \Delta', y : B$. In the first case we know that $\Gamma \vdash_f A : s$ by assumption of the rule. In the second case we know that $\Gamma, x : A, \Delta'$ is also legal by Lemma 5.3, hence the statement follows from the IH. If the last rule was any other rule, the context in the conclusion of the rule also appears in one of the assumptions, hence the statement follows from the IH. □

The following lemma states some basic properties about the equality judgement. The first part is that a convertibility proof determines the corresponding convertible terms, the second part states that every term in an equality judgement has a type. We will later see that these types need not be the same (cf. Example 7.6).

Lemma 5.8.

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. First note that the last steps in the derivations of $\Gamma \vdash_f H : A = B$ and $\Gamma \vdash_f H : \tilde{A} = \tilde{B}$ must be the same. Now we use induction on these derivations, and distinguish cases according to the last step used.

If the last step used was (ref), (beta), or (iota) then the term H uniquely determines A and B. For example with the iota rule, then $H \equiv \iota(a^{H'})$ for some a, H'. Then $A \equiv a \equiv \tilde{A}$ and $B \equiv a^H \equiv \tilde{B}$.

If the last step used was (sym), (trans) or (app-eq) the statements follow trivially from the IH. For example in the case (app-eq) then $H \equiv H'H''$ such that $\Gamma \vdash_f H' : F = F'$ and $\Gamma \vdash_f H'' : a = a'$. By the induction hypothesis these F, F', a, a' are uniquely determined, so Fa and $F'a'$ are so, too.

The last two cases, (prod-eq) and (abs-eq) are less trivial. We treat (prod-eq), the rule (abs-eq) is similar. Now $H \equiv \{H', [x : A']H''\}$ and $A \equiv \Pi x{:}A'.B'$, $B \equiv \Pi x'{:}A''.B''$, $\tilde{A} \equiv \Pi x{:}\tilde{A}'.\tilde{B}'$ and $\tilde{B} \equiv \Pi x'{:}\tilde{A}''.\tilde{B}''$. Then we know by the IH that $A' \equiv \tilde{A}$, $A'' \equiv \tilde{A}'$, $B' \equiv \tilde{B}$ and $B''[x' := x^{H'}] \equiv \tilde{B}'[x' := x^{H'}]$. We want to show that $B'' \equiv \tilde{B}'$ and for this we assume that $x \not\equiv x'$ (which can be done using Lemma 5.2). Since $\Gamma, x : A$ is legal, x does not occur in Γ and hence also not in $\Gamma, x' : C$ for any C. Since one of the assumptions of the (prod-eq) rule is $\Gamma, x' : A'' \vdash_f B'' : s$ and $\Gamma, x' : \tilde{A}' \vdash_f \tilde{B}' : s$ we know that x does not occur in $\tilde{B}'$ or B''. This means that again all occurrences of x in $B''[x' := x^{H'}]$ or $\tilde{B}'[x' := x^{H'}]$ came from the substitution, so if we substitute x' back, we obtain $B'' \equiv \tilde{B}'$.

2. We use induction on the used derivation. If the last used rule is (ref), then it follows by the assumption of the rule. If the last rule was (sym) or (trans) then it follows by the IH. If the last rule was (beta), $\dfrac{\Gamma \vdash_f a : A : s_1 \quad \Gamma, x : A \vdash_f b : B : s_2}{\Gamma \vdash_f \beta((\lambda x{:}A.b)a) : (\lambda x{:}A.b)a = b[x := a]}\ (s_1, s_2, s_3) \in \mathcal{R}$, then $\Gamma \vdash_f (\lambda x{:}A.b)a : B[x := a]$ by applying (abs) and (app) and $\Gamma \vdash_f b[x := a] : B[x := a]$ by Substitution (Lemma 5.6). The statement follows for (prod-eq), (abs-eq), (app-eq) and (iota) by applying (prod), (abs), (app) and (conv), respectively.

□

The following Lemma is also common in PTSs. It states that most types can be typed themselves by a sort. The only exception to this is so-called ‘top-sorts’, which are sorts s for which there doesn’t exist a sort t such that $(s, t) \in \mathcal{A}$. In this case, s cannot have a type under any context, hence the distinction in the following Lemma.

Lemma 5.9 (Type Correctness). `[TypeCorrect]` *If B is a Γ-type then B is a Γ-semitype. This means that if $\Gamma \vdash_f A : B$ then there is a sort s such that either $B \equiv s$ or $\Gamma \vdash_f B : s$.*

Proof. We use induction on the derivation of $\Gamma \vdash_f A : B$. If the last rule was (sort), (prod) or (conv), then the statement follows immediately from hypotheses in the rule. If the last rule was (var) then it follows by applying Context Generation (Lemma 5.7) and Thinning (Lemma 5.5). If the last rule was (abs), then the statement follows by applying the rule (prod).

If the last rule was (app), then there are F, a, A', B' such that $A \equiv Fa$ and $B \equiv B'[x := a]$ and $\Gamma \vdash_f F : \Pi x{:}A'.B'$, $\Gamma \vdash_f a : A'$. By the IH on $\Gamma \vdash_f F : \Pi x{:}A'.B'$ we know that $\Gamma \vdash_f \Pi x{:}A'.B' : s'$ (since $\Pi x{:}A'.B' \not\equiv t$ for any sort t). The last rule using in the derivation of this judgement must have been (prod), so we know by the assumptions of (prod) that $\Gamma, x : A' \vdash_f B' : s$ for some s. By the substitution lemma 5.6 the statement $\Gamma \vdash_f B : s$ follows. □

5.3 Uniqueness of Derivations

In this section we will prove Theorem 4.1. We will prove this by induction on the derivation, as always, but we first 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 5.10. We call two terms A and B *comparable* if either $A \equiv B$ or there is an $n \geq 0$, there are n terms $A_1, \ldots, A_n \in \mathcal{T}$ and two sorts s, t such that

1. we have $A \equiv \Pi x_1{:}A_1.\Pi x_2{:}A_2.\cdots \Pi x_n{:}A_n.s$;
2. we have $B \equiv \Pi x_1{:}A_1.\Pi x_2{:}A_2.\cdots \Pi x_n{:}A_n.t$.

Note that in particular any two sorts are comparable and that A and B are comparable iff $\Pi x{:}C.A$ and $\Pi x{:}C.B$ are comparable.

Lemma 5.11. *If $\Gamma \vdash_f M : A$ and $\Gamma \vdash_f M : B$ then A and B are comparable.*

Proof. We use induction on the structure of M. Note that in each case there is a unique rule which could have been used to conclude $\Gamma \vdash_f M : A$ and $\Gamma \vdash_f M : B$.

If M is a sort, then the last used rule for the derivations is (sort), so A and B are both sorts, hence comparable. If $M \equiv x$ is a variable, then the last used rule is (var), and then we know that $(x : C) \in \Gamma$ for exactly one C. This means that $A \equiv C \equiv B$.

If M is a product, then the last rule is (prod), and both A and B are sorts again.

If $M \equiv \lambda x{:}C.b$ is an abstraction, then the last rule is (abs). This means that the judgements were concluded from $\Gamma, x : C \vdash_f b : B_1$ resp. $\Gamma, x : C \vdash_f b : B_2$ for some B_1, B_2. By the IH we conclude that B_1 and B_2 are comparable. Now $A \equiv \Pi x{:}C.B_1$ and $B \equiv \Pi x{:}C.B_2$, so A and B are also comparable.

If $M \equiv N^H$ is a conversion, then H uniquely determines the type of M by Lemma 5.8.1, so $A \equiv B$. □

We can now prove Theorem 4.1, we will actually prove a little stronger result. For the definition of derivation tree, see Remark 4.2.

Theorem 5.12 (Uniqueness of Derivation). `[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 $\Gamma \vdash_f H : M' = N'$*

Note that in the second statement A is not required to be equal to A'.

Proof. We will prove these statements by simultaneous induction on the derivation of the first judgement in each item, distinguishing cases according to the last applied rule. Most cases are easy, so we only treat a few of them.

If the last rule was (sort), then it was concluded from $\Gamma \vdash_f$. By the IH we know that any two derivation trees of this judgement are equal. The derivation trees of the judgement after applying (sort) is this tree with one node with label (sort) added. Hence the derivation trees are equal.

If the last rule was (prod), then $M \equiv \Pi x{:}C.B$ and $M' \equiv \Pi x{:}C.B'$ where B is comparable to B'. Now the last rule of $\Gamma \vdash_f M' : A'$ was also (prod). We can apply the IH to both hypotheses (note that the corresponding contexts are equal), to obtain that the derivation trees to these judgements is equal. We conclude that the full derivation trees are also equal.

If the last rule was (abs), then M and M' are both abstractions, and then they must be equal, say $\lambda x{:}A.b$. This also determines A and b. Now by Lemma 5.11 we conclude that the types B and B' of b used in the hypothesis of the rules are comparable. We can now apply the IH to all three hypotheses to note that the derivation trees are equal, finishing this case.

If the last used rule was (conv), then we also have that $M \equiv M'$, equal to say a^H. Now this H uniquely determines the type A' of a^H by Lemma 5.8.1. This means that we can apply the IH to all hypotheses to the rule, to finish this case.

All other rules are similar to or easier than the rules we treated. For all equality rules, one first has to use Lemma 5.8.1. □

With this proof we have also proven Theorem 4.1.

5.4 Erasure map

In this section we will prove some properties about the erasure map as defined in Definition 4.3.7. We start with the implication $\lambda_f\mathbf{S} \Rightarrow \lambda\mathbf{S}$.

Theorem 5.13. `[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 and follow immediately from an assumption or by induction hypothesis and applying the same rule. □

Lemma 5.14. *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. We only prove the second statement, the other two are proven similarly. By Lemma 5.2 we can enforce that $x \neq x'$. Note that $\Gamma, x' : A', \Delta$ is legal by Lemma 5.3, so by Context Generation (Lemma 5.7) we know that $\Gamma \vdash_f A' : t$ for some sort t. By Weakening (Lemma 5.4) we conclude $\Gamma, x : A \vdash_f A' : t$. Also, $\Gamma, x : A \vdash_f x : A$ by (var) and by the Weakening Lemma we conclude $\Gamma, x : A \vdash_f H : A = A'$. Hence by (conv), $\Gamma, x : A \vdash_f x^H : A'$. By Weakening again we also conclude $\Gamma, x : A, x' : A', \Delta \vdash_f M : N$, so by Substitution (Lemma 5.6) the result follows. □

Proposition 5.15 (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

$$\forall A', \Gamma\colon \text{if } A \text{ and } A' \text{ have types under } \Gamma \text{ and } |A| \equiv |A'|, \text{ then } \Gamma \vdash_f A = A'.$$

In every step we use induction on the structure of A'.

If $A' = C^{H'}$, then we can use the induction hypothesis for $(A, A') = (A, C)$, since $|A| \equiv |A'| \equiv |C|$. This gives us a convertibility proof H'' such that $\Gamma \vdash_f H'' : A = C$. Now $A \equiv C^{H'}$ has a type under Γ which must be concluded from (conv), so by the assumptions of (conv) and applying (iota) we know that $\Gamma \vdash_f \iota(A') : C = A'$. By (trans) we conclude $\Gamma \vdash_f H : A = A'$ for $H \equiv H'' \cdot \iota(A')$. If $A = C^{H'}$, then we can use the induction hypothesis on $(A, A') = (C, A')$, and then the rest of the case is similar.

If neither A nor A' is of the form $C^{H'}$ then A and A' have the same outer structure since $|A| \equiv |A'|$. We now treat the other cases:

If A and A' are sorts or variables, then $|A| \equiv |A'|$ implies $A \equiv A'$. Since A has a type under Γ, we know by (ref) that $\Gamma \vdash_f \overline{A} : A = A'$.

If A and A' are products, say $A \equiv \Pi x{:}B.C$ and $A' \equiv \Pi x'{:}B'.C'$ then $|B| \equiv |B'|$ and $|C| \equiv |C'|[x' := x]$. The last rule in the derivation of the judgements where A and A' are typed under Γ can only be (prod), concluded from

$$\begin{array}{ll} \Gamma \vdash_f B : s_1 & \qquad\qquad \Gamma, x : B \vdash_f C : s_2 \\ \Gamma \vdash_f B' : s_1' & \qquad\qquad \Gamma, x' : B' \vdash_f C' : s_2' \end{array}$$

and by the IH on (B, B') there is an H_1 such that $\Gamma \vdash_f H_1 : B = B$ and hence by Lemma 5.14 we conclude that $\Gamma, x : B \vdash_f C'[x' := x^{H_1}] : s_2'$. So C and $C'[x' := x^{H_1}]$ have a type under $\Gamma, x : B$ and $|C'[x' := x^{H_1}]| \equiv |C'|[x' := |x^{H_1}|] \equiv |C'|[x' := x] \equiv |C|$. Hence by the IH on $(C, C'[x' := x^{H_1}])$ we conclude that there is an H_2 such that $\Gamma, x : B \vdash_f H_2 : C = C'[x' := x^{H_1}]$. By (prod-eq) we conclude that $\Gamma \vdash_f H : A = A'$ for $H = \{H_1, [x : B]H_2\}$.

If A and A' are abstractions, we can apply an argument similar to the product-case.

If A and A' are both applications, say $A \equiv Fa$ and $A' \equiv F'a'$ then the last rule was (app), hence by the assumptions of (app) and by induction hypothesis we can check all assumptions for (app-eq), from which the statement follows. ☐

Lemma 5.16.

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_2 of $|a_1|$ and a lift A_3 of $|A_1|$ such that $\Gamma \vdash_f a_2 : A_3 : s$.*
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. 1. We use induction on the structure on A. If A is a sort, then $A \equiv s$, and since A has a type under Γ, we can apply (refl). By $|A| \equiv s$ we conclude that A cannot be a variable, product, abstraction or application. The only case left is A is a conversion, say $A \equiv C^{H'}$. Since A has type under Γ, which must have been derived using (conv) as last rule, we know by the assumptions to (conv) we know that there exist M, N such that $\Gamma \vdash_f C : M$, $\Gamma \vdash_f N : s$ and $\Gamma \vdash_f H' : M = N$. Now by (iota) we know that $\Gamma \vdash_f \iota(A) : C = A$. Because $|C| \equiv |A| \equiv s$, we conclude by the IH that $C \equiv s$ or $\Gamma \vdash_f H'' : C = s$ for some H''. Then $\Gamma \vdash_f H : A = s$ follows, in the first case for $H \equiv \iota(B)^\dagger$ and in the second case for $H \equiv \iota(B)^\dagger \cdot H''$.

2. By Lemma 5.9 we know that A also is a Γ-semitype. If A and B are both sorts, then they must be the same sort, and we can take $b \equiv a$. Otherwise we claim that $\Gamma \vdash H : A = B$ for some H. If A and B both have a type under Γ, then this follows by Erasure Injectivity (Proposition 5.15). If one of them has a type under Γ and the other is a sort, it follows from Part 1 of this Lemma. Now take $b \equiv a^H$. Then $|a| \equiv |b|$ and by (conv) we conclude that $\Gamma \vdash_f b : B$.

3. Apply part 2 twice, first to find A_3 then to find a_2.

4. Apply part 2 twice to find lifts b_1 and b_2 satisfying $\Gamma \vdash_f b_1 : B$ and $\Gamma \vdash_f b_2 : B$. By Erasure Injectivity (Proposition 5.15) we conclude that $\Gamma \vdash_f b_1 = a_1$ and $\Gamma \vdash_f a_2 = b_2$. We conclude that $\Gamma \vdash_f b_1 = b_2$ by applying (trans) twice. □

5.5 Equality of substitutions

We are almost ready to prove the implication $\lambda_e\mathbf{S} \Rightarrow \lambda_f\mathbf{S}$. In fact, we can do all cases of the induction, except the (app-eq)-case. For that case we need one more Lemma, which was tricky to prove. 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 5.20. This cannot be done by a simple induction on the first judgement. To prove the Corollary we need a more general statement (Proposition 5.19), where $x : T$ can occur anywhere in the context.

To see why this is the case, let us try to prove Corollary 5.20. So we want to conclude $\Gamma \vdash_f M[x := a_1] = M[x := a_2]$ from the statements

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

The obvious way to do this is by induction on either M or induction on the derivation of the judgement $\Gamma, x : T \vdash_f M : N$. These inductions are practically the same thing. So when we do either induction, we have a problem in the product or abstraction case. In the product case $\Gamma, x : T \vdash_f \Pi y{:}A.B : s_3$ 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 when 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 when 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 $A[x := a_2]$. But this gives a problem, because there seems no good context Δ to the judgement $\Delta \vdash_f M[x := a_1] = M[x := a_2]$. 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 when 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

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

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 when 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

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

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

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 $\mathrm{dom}(\Delta)$. 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 when 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 5.17. For a vector $\vec{x} = (x_1, \ldots, x_n)$ write $(\vec{x}, x) := (x_1, \ldots, x_n, x)$, $\vec{x}_i := (x_1, \ldots, x_i)$ and $\vec{x}^i := (x_i, \ldots, x_n)$. Also for a pseudocontext $\Delta \equiv y_1 : D_1, \ldots, y_n : D_n$ write $\Delta_i :\equiv y_1 : D_1, \ldots, y_i : D_i$ (which is the empty context for $i = 0$) and $\Delta^i :\equiv y_i : D_i, \ldots, y_n : D_n$ (which is empty for $i = n + 1$).

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 5.14 repeatedly.

Lemma 5.18. *Let Γ and $\Delta \equiv y_1 : D_1, \ldots, y_n : D_n$ and $\Delta' \equiv y'_1 : D'_1, \ldots, y'_n : D'_n$ be pseudocontexts 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. We prove the statements separately by induction on k. For $k = 0$ it is the assumption, for the induction step use Lemma 5.7 to derive $\Gamma, \Delta_{i-1} \vdash_f D_i : s_i$ and then apply Lemma 5.14 to the induction hypothesis to complete the induction step. □

Now we can prove the general statement.

Proposition 5.19. [equality_subst_ext] *Let* Γ *and* $\Delta \equiv y_1 : D_1, \dots, y_n : D_n$ *be pseudocontexts. Suppose that the following judgements hold.*

- $\Gamma \vdash_f a_1 = a_2$;
- $\Gamma \vdash_f a_1 : T$;
- $\Gamma \vdash_f a_2 : T$;
- $\Gamma, x : T, \Delta \vdash_f M : N$;
- *for all* $i \in \{1, \dots, n\}$ *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 $\Gamma, x : T, \Delta \vdash_f M : N$.

If the last rule is (sort) with conclusion $\Gamma, x : T, \Delta \vdash_f s_1 : s_2$ then by Substitution (Lemma 5.6) we know that $\Gamma, \Delta[x := a_1]$ is legal. We conclude $\Gamma, \Delta[x := a_1] \vdash_f \overline{s_1} : s_1 = s_1$, as desired.

If the last rule is (var) with conclusion $\Gamma, x : T, \Delta \vdash_f z : N$, note that the context $\Gamma, \Delta[x := a_1]$ is legal. We distinguish three cases.

- If $(z : N) \in \Gamma$ then we conclude $\Gamma, \Delta[x := a_1] \vdash_f z = z$ by (var) and (ref), finishing this case.
- If $z = x$ then $N \equiv T$. Now we conclude $\Gamma, \Delta[x := a_1] \vdash_f a_1 = a_2$ by Thinning (Lemma 5.5) and we are done since y_i does not occur in a_2 by Lemma.
- If $(z : N) \in \Delta$ then $N \equiv D_k$ for some k and we need to prove that $\Gamma, \Delta[x := a_1] \vdash_f z = z^{H_k}$ (note that H_k does not contain y_i as free variable for $i \geq k$). By Thinning we observe that

$$\Gamma, \Delta[x := a_1] \vdash_f H_k : D_k[x := a_1] = D_k[x := a_2][\vec{y}_{k-1} := \vec{y}_{k-1}^{\vec{H}_{k-1}}].$$

We also have $\Gamma, \Delta[x := a_1] \vdash_f z : D_k[x := a_1]$ by (var) and $\Gamma, \Delta[x := a_2] \vdash_f D_k[x := a_2] : s$ by Context Generation (Lemma 5.7) and Thinning. Then by Lemma 5.18 we conclude that

$$\Gamma, \Delta[x := a_1] \vdash_f D_k[x := a_1][\vec{y}_{k-1} := \vec{y}_{k-1}^{\vec{H}_{k-1}}] : s,$$

noting that y_i does not appear in D_k for $i \geq k$. We conclude by applying (iota).

If the last rule is (prod), the judgement was concluded from $\Gamma, x : T, \Delta \vdash_f A : s_1$ and $\Gamma, x : T, \Delta, y : A \vdash_f B : s_2$. By induction hypothesis on the first judgement we get $\Gamma, \Delta[x := a_1] \vdash_f H : A[x := a_1] = A[x := a_2][\vec{y} := \vec{y}^{\vec{H}}]$. Using substitution we obtain

$$\begin{array}{ll} \Gamma, \Delta[x := a_1] \vdash_f A[x := a_1] : s_1; & \Gamma, \Delta[x := a_1], y : A[x := a_1] \vdash_f B[x := a_1] : s_2; \\ \Gamma, \Delta[x := a_2] \vdash_f A[x := a_2] : s_1; & \Gamma, \Delta[x := a_2], y : A[x := a_2] \vdash_f B[x := a_2] : s_2. \end{array}$$

By application of Lemma 5.18 on the bottom two judgements we obtain

$$\Gamma, \Delta[x := a_1] \vdash_f A[x := a_2][\vec{y} := \vec{y}^{\vec{H}}] : s_1$$

and

$$\Gamma, \Delta[x := a_1], y : A[x := a_2][\vec{y} := \vec{y}^{\vec{H}}] \vdash_f B[x := a_2][\vec{y} := \vec{y}^{\vec{H}}] : s_2$$

and by using the induction hypothesis on the other judgement (using $D_{n+1} \equiv A$ and $H_{n+1} \equiv H$) we obtain

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

We finish this case using (prod-eq).

The case (abs) is similar to (prod) and (app) is easier.

If the last rule was (conv), the judgement was concluded from $\Gamma, x : T, \Delta \vdash_f a : A$ and $\Gamma, x : T, \Delta \vdash_f A' : s$ and $\Gamma, x : T, \Delta \vdash_f H : A = A'$. By induction hypothesis on the first judgement we obtain $\Gamma, \Delta[x := a_1] \vdash_f a[x := a_1] = a[x := a_2][\vec{y} := \vec{y}^{\vec{H}}]$. Using substitution on the three judgements, we obtain using (iota) and (sym) the judgements $\Gamma, \Delta[x := a_1] \vdash_f a^H[x := a_1] = a[x := a_1]$ and $\Gamma, \Delta[x := a_2] \vdash_f a[x := a_2] = a^H[x := a_2]$. On the second judgement we can use the previous Lemma to obtain $\Gamma, \Delta[x := a_1] \vdash_f a[x := a_2][\vec{y} := \vec{y}^{\vec{H}}] = a^H[x := a_2][\vec{y} := \vec{y}^{\vec{H}}]$. Using (trans) twice, we obtain the desired judgement. □

Corollary 5.20. `[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 5.19. □

5.6 Equivalence

Theorem 5.21. `[PTSeq2PTSF]` *For all $\lambda_e\boldsymbol{S}$ contexts Γ, and all $\lambda_e\boldsymbol{S}$ 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.

(nil) Obvious.

(cons) If the last applied rule was $\dfrac{\Gamma \vdash_e A : s}{\Gamma, x : A \vdash_e}$ $x \notin \operatorname{dom}\Gamma$ then by the IH there exists a legal lift Γ' of Γ. Then also by the IH there are lifts A'' and T' of A and s respectively such that $\Gamma' \vdash_f A'' : T'$. By Lemma 5.16.2 there is a lift A' of A satisfying $\Gamma' \vdash_f A' : s$. Now $\Gamma', x : A'$ is a legal lift of $\Gamma, x : A$, as desired.

For all other rules, the existence of a legal lift immediately follows from the IH in every induction step. Let Γ' be a legal lift of Γ, where Γ is the context of the conclusion of the particular rule.

(sort) If the last applied rule was $\dfrac{\Gamma \vdash_e}{\Gamma \vdash_e s_1 : s_2}$ $(s_1, s_2) \in \mathcal{A}$ then we know by (sort) that $\Gamma' \vdash_f s_1 : s_2$.

(var) If the last applied rule was $\dfrac{\Gamma \vdash_e}{\Gamma \vdash_e x : A}$ $(x : A) \in \Gamma$ then we know that $(x : A') \in \Gamma'$ for some lift A' of A, hence by (var) we conclude $\Gamma' \vdash_f x : A'$.

(prod) If the last applied rule was $\dfrac{\Gamma \vdash_e A : s_1 \quad \Gamma, x : A \vdash_e B : s_2}{\Gamma \vdash_e \Pi x{:}A.B : s_3}$ $(s_1, s_2, s_3) \in \mathcal{R}$ then by the IH there are lifts A'', T' of A, s_1 respectively such that $\Gamma' \vdash_f A'' : T'$. By Lemma 5.16.2 we conclude that there is a lift A' of A such that $\Gamma' \vdash_f A' : s_1$. So $\Gamma', x : A'$ is legal and by applying the IH and the same lemma again we conclude $\Gamma', x : A' \vdash_f B' : s_2$ for some lift B' of B. We finish by applying (prod).

(abs) If the last applied rule was $\dfrac{\Gamma \vdash_e A : s_1 \quad \Gamma, x : A \vdash_e b : B : s_2}{\Gamma \vdash_e \lambda x{:}A.b : \Pi x{:}A.B}$ $(s_1, s_2, s_3) \in \mathcal{R}$ then we see by the IH and that Lemma 5.16.2 that $\Gamma' \vdash_f A' : s_1$ for some lift A' of A, hence that $\Gamma', x : A'$ is legal. Now $\Gamma', x : A' \vdash_f b_1 : B_1$ and $\Gamma', x : A' \vdash_f B_2 : M$ for lifts b_1, B_1, B_2, M of b, B, B, s_2 respectively. By Lemma 5.16.3 we conclude that there are lifts b', B' of b, B respectively such that $\Gamma', x : A' \vdash_f b' : B' : s_2$. We finish by applying (abs).

(app) If the last applied rule was $\dfrac{\Gamma \vdash_e F : \Pi x{:}A.B \quad \Gamma \vdash_e a : A}{\Gamma \vdash_e Fa : B[x := a]}$ then by the IH twice we find that $\Gamma' \vdash_f F'' : T'$ and $\Gamma' \vdash_f a'' : A''$ for lifts F'', T', a'', A'' of F, $\Pi x{:}A.B$, a, A respectively. Since we know that $|T'| = \Pi x{:}A.B$, we can write $T' \equiv (\Pi x{:}A'.B')^{H_1\, H_2\, \cdots\, H_n}$ for lifts A', B' of A, B respectively. Then by Lemma 5.9 we know that T' has a type under Γ. Now as long as $n > 0$ we know that the only way to type T' is using (conv), and we know that when we remove the outermost convertibility proof, we also get a term which has a type under Γ. By induction we conclude that $\Pi x{:}A'.B'$ has a type under Γ. Since the only way to type a product is using (prod), we conclude that this product is typed by a sort and that $\Gamma \vdash_f A' : s$ for some sort s. By applying Lemma 5.16.2 twice we find lifts a' and F' of a and F respectively such that $\Gamma' \vdash_f a' : A'$ and $\Gamma' \vdash_f F' : \Pi x{:}A'.B'$. By (app) we conclude that $\Gamma' \vdash_f F'a' : B'[x := a']$, as desired.

(conv) If the last applied rule was $\dfrac{\Gamma \vdash_e a : A \quad \Gamma \vdash_e A = A' : s}{\Gamma \vdash_e a : A'}$ then by the IH there are lifts A_1, A_1', M of A, A', s respectively such that $\Gamma' \vdash_f A_1 = A_1'$, $\Gamma' \vdash_f A_1 : M$ and $\Gamma' \vdash_f A_1' : M$. By applying Lemma 5.16.4 we find lifts A_2, A_2' of A, A' respectively such that $\Gamma' \vdash_f A_2 : s$ and $\Gamma' \vdash_f A_2' : s$ and $\Gamma' \vdash_f A_2 = A_2'$. Now by the IH and Lemma 5.16.2 we find a lift a' of a such that $\Gamma' \vdash_f a' : A_2$. The conclusion follows by applying (conv).

(ref) If the last applied rule was $\dfrac{\Gamma \vdash_e A : B}{\Gamma \vdash_e A = A : B}$ then by the IH we find A', B' such that $\Gamma' \vdash_f A' : B'$. The equality judgement follows by applying (ref).

(sym) If the last applied rule was (sym), the result follows immediately from the IH and (sym).

(trans) If the last applied rule was (trans), the result follows immediately from the IH and (trans).

(beta) If the last applied rule was $\dfrac{\Gamma \vdash_e a : A : s_1 \quad \Gamma, x : A \vdash_e b : B : s_2}{\Gamma \vdash_e (\lambda x{:}A.b)a = b[x := a] : B[x := a]}$ $(s_1, s_2, s_3) \in \mathcal{R}$ then by the IH twice and Lemma 5.16.3 we find lifts a', A' of a, A respectively, such that $\Gamma' \vdash_f a' : A' : s_1$. Then $\Gamma', x : A'$ is a legal context, hence by the IH twice and Lemma 5.16.3 again we find lifts b', B' of b, B such that $\Gamma', x : A' \vdash_f b' : B' : s_2$. The equality judgement follows from (beta) and the typing judgements from (the proof of) Lemma 5.8.2.

(prod-eq) If the last applied rule was $\dfrac{\Gamma \vdash_e A = A' : s_1 \qquad \Gamma, x : A \vdash_e B = B' : s_2}{\Gamma \vdash_e \Pi x{:}A.B = \Pi x{:}A'.B' : s_3}$ $(s_1, s_2, s_3) \in \mathcal{R}$ then by the IH and Lemma 5.16.4 we find a convertibility proof H and lifts A_1, A_1' of A, A' respectively such that $\Gamma' \vdash_f A_1 : s_1$ and $\Gamma' \vdash_f A_1' : s_1$ and $\Gamma' \vdash_f H : A_1 = A_1'$. Then $\Gamma', x : A_1$ is legal, so by the IH and Lemma 5.16.4 again we find lifts B_1, B_2' of B, B' such that $\Gamma', x : A_1 \vdash_f B_1 : s_2$ and $\Gamma', x : A_1 \vdash_f B_2' : s_2$ and $\Gamma', x : A_1 \vdash_f B_1 = B_2'$. Define $B_1' :\equiv B_2'[x := x'^{H^\dagger}]$ for a fresh variable x'. Then by Lemma 5.14 we conclude that $\Gamma, x' : A_1' \vdash_f B_1' : s_2$. Now note that $|B_1'[x' := x^H]| \equiv |B_2'[x := x'][x' := x]| \equiv |B_2'|$ and that by Lemma 5.14 again we find $\Gamma, x : A \vdash_f |B_1'[x' := x^H]| : s_2$. Hence by Erasure Injectivity (Proposition 5.15) we conclude that $\Gamma, x : A_1 \vdash_f B_2' = B_1'[x' := x^H]$, so by (trans) we obtain $\Gamma, x : A_1 \vdash_f B_1 = B_1'[x' := x^H]$. The result follows by using (prod) twice and (prod-eq) once.

(abs-eq) The step (abs-eq) follows similar to the step (prod-eq).

(app-eq) If the last applied rule was $\dfrac{\Gamma \vdash_e F = F' : \Pi x{:}A.B \qquad \Gamma \vdash_e a = a' : A}{\Gamma \vdash_e Fa = F'a' : B[x := a]}$ then by the IH we find lifts F_1, F_1' and T of F, F' and $\Pi x : A.B$ respectively such that $\Gamma' \vdash_f F_1 : T$ and $\Gamma' \vdash_f F_2 : T$ and $\Gamma' \vdash_f F_1 = F_1'$. Using the same trick as in the step (app), we find lifts A' and B' of A and B respectively, such that $\Gamma' \vdash_f A' : s_1$ and $\Gamma', x : A' \vdash_f B' : s_2$ and $\Gamma' \vdash_f \Pi x{:}A'.B' : s_3$ for some $(s_1, s_2, s_3) \in \mathcal{R}$. Now by the IH once more and Lemma 5.16.4 twice we find lifts F_2, F_2', a_1, a_1' of F, F', a, a' respectively such that

$$\begin{array}{lll} \Gamma' \vdash_f F_2 = F_2'; & \Gamma' \vdash_f F_2 : \Pi x{:}A'.B'; & \Gamma' \vdash_f F_2' : \Pi x{:}A'.B'; \\ \Gamma' \vdash_f a_1 = a_1'; & \Gamma' \vdash_f a_1 : A'; & \Gamma' \vdash_f a_1' : A'. \end{array}$$

Then $\Gamma' \vdash_f F_2 a_1 = F_2' a_1'$ follows by (app-eq) and by (app) twice we conclude $\Gamma \vdash F_2 a_1 : B'[x := a_1]$ and $\Gamma \vdash F_2' a_1' : B'[x := a_1']$. To show that we can find terms with equal type, we use Corollary 5.20 to find H such that $\Gamma' \vdash_f H : B'[x := a_1] = B'[x := a_1']$. Now we define $M = (F_2 a_1)^H$. By Substitution we know that $\Gamma' \vdash_f B'[x := a_1'] : s_2$, and by (conv) we now conclude $\Gamma' \vdash_f M : B'[x := a_1']$ and by (iota) and (trans) we conclude $\Gamma' \vdash_f M = F_2' a_1'$, finishing this part.

(conv-eq) If the last applied rule was $\dfrac{\Gamma \vdash_e a = a' : A \quad \Gamma \vdash_e A = A' : s}{\Gamma \vdash_e a = a' : A'}$ then by the IH and Lemma 5.16.4 we find a convertibility proof H and lifts A_1 and A_1' of A and A' respectively such that $\Gamma' \vdash_f H : A_1 = A_1'$ and $\Gamma' \vdash_f A_1 : s$ and $\Gamma' \vdash_f A_1' : s$. By the IH and Lemma 5.16.4 again we find lifts a_1, a_1' of a, a' respectively such that $\Gamma' \vdash_f H : a_1 = a_1'$ and $\Gamma' \vdash_f a_1 : A_1$ and $\Gamma' \vdash_f a_1' : A_1$. Define $a_2 :\equiv a_1^H$ and $a_2' :\equiv a_2^H$. Then $\Gamma' \vdash_f a_2 : A_1'$ and we conclude $\Gamma' \vdash_f a_2' : A_1'$ by (conv) and $\Gamma' \vdash_f a_2 = a_2'$ by (iota) twice, (sym) once and (trans) twice.

This finishes the proof. □

Theorem 5.22. [PTS12PTSF] *For all $\lambda\boldsymbol{S}$-contexts Γ and all $\lambda\boldsymbol{S}$-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. The first two implications follow by Theorem 3.1 and Theorem 5.21. For the third implication we need some more work, because in $\lambda_e\mathbf{S}$, equalities are only allowed between terms with equal types. But we can still prove it using the Church-Rosser Theorem and Subject Reduction.

By Theorem 2.1.2 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 (Theorem 2.9) 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 5.21 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 (Proposition 5.15) we conclude that $\Gamma' \vdash_f C' = C''$. We conclude that $\Gamma' \vdash_f A' = B'$ using (trans) twice. □

Now we can prove the equivalence between $\lambda\mathbf{S}$ and $\lambda_f\mathbf{S}$.

Theorem 5.23 (Equivalence between $\lambda\mathbf{S}$ and $\lambda_f\mathbf{S}$). `[PTSlequivPTSF]` *For all $\lambda\boldsymbol{S}$-contexts Γ and all $\lambda\boldsymbol{S}$-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 $\Gamma' \vdash_f H : A' = B'$.*

Proof. All implications from lift to right follow from Theorem 5.22. All implications in the other direction follow from Theorem 5.13. For the third implication we also need Lemma 5.8.2. □

Corollary 5.24 (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 5.23 we conclude that $\Pi x{:}|A|.|B| \simeq_\beta \Pi x{:}|A'|.|B'|$. Using Church-Rosser, we obtain a common beta-reduct, which must also be a product, say $\Pi x{:}A''.B''$. Then $|A|, |A'| \twoheadrightarrow_\beta A''$ and $|B|, |B'| \twoheadrightarrow_\beta B''$, so $|A| \simeq_\beta |A'|$ and $|B| \simeq_\beta |B'|$. Also, by Equality Typing (Lemma 5.8.2) both $\Pi x{:}A.B$ and $\Pi x{:}A'.B'$ have a type under Γ, and the last used rule in the derivations of these judgements must be (prod). Hence the following judgements hold.

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

We start with the convertibility between A and A'. By Theorem 5.23 we conclude that $|\Gamma| \vdash |A| : s_1$ and $|\Gamma| \vdash |A'| : s_1'$, hence by Theorem 5.22 there are lifts A_1 and A_1' of $|A|$ and $|A'|$ respectively, such that $\Gamma \vdash_f A_1 = A_1'$, and from Equality Typing we conclude that A_1 and A_1' both have a type under Γ. Now by Erasure Injectivity (Proposition 5.15) twice, 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 $\Gamma \vdash H : A = A'$.

Now we prove the convertibility between B and $B'[x := x^H]$. By Lemma 5.14 we see that $\Gamma, x : A \vdash_f B'[x := x^H] : s_2'$ and we already concluded that $\Gamma, x : A \vdash_f B : s_2$. By Theorem 5.23 we now know that $|\Gamma, x : A| \vdash |B| : s_2$ and $|\Gamma, x : A| \vdash |B'| : s_2'$ using that $|B'[x := x^H]| \equiv |B'|$. By Theorem 5.22 there are lifts B_1 and B_1' of $|B|$ and $|B'|$ respectively, such that $\Gamma, x : A \vdash_f B_1 = B_1'$. Using the same steps as before with Equality Typing and Erasure Injectivity, we conclude that $\Gamma, x : A \vdash_f B = B'[x := x^H]$. This completes the proof. □

6 Formalisation

The proofs in the previous section are rather technical, and even though we wrote most proofs in a rather detailed way, in almost every proof we still skipped some steps. To give more confidence in the proofs and make sure we did not make any mistakes we have completely formalised all proofs. Actually, during the formalisation we found quite some mistakes, and some were highly nontrivial to fix. The gravest mistake was that in a earlier version of Theorem 5.21 in the (app-eq)-step, the need to prove $\Gamma' \vdash_f B'[x := a_1] = B'[x := a_1']$ was overlooked, which means that the theory of Section 5.5 was developed only after discovering the mistake. This illustrates one of the merits of formalising proofs: in long proofs there will almost always be some mistakes, and they will be discovered when formalising the proof.

We used the proof assistant Coq [The Coq development team, 2012] (version 8.4) to verify all proofs in this thesis. As starting point we used the formalisation of Siles [Siles and Herbelin, 2012], who has formalised his proof of Theorem 3.1 in Coq. In his formalisation he defined three typing systems PTS, PTS$_e$ and PTS$_{atr}$, where the last one was used to prove the equivalence between the first two. We could mimic the definition and the proofs of the basic properties of these systems for PTS$_f$, which was really helpful. For the more advanced Lemmas and Theorems (Section 5.3 and later) we still had to do everything ourselves.

The formalisation follows the thesis quite well. Theorems in this thesis which have been formalised are annotated with the name of the result in the Coq formalisation, using the notation `[Coq name]`. There are a few differences between the formalisation of the proofs and the proofs presented in this paper, though. The most important difference is that we used a different way of writing variables in terms, namely via *de Bruijn indices* [de Bruijn, 1972]. When writing a bound variable with de Bruijn indices, instead of writing the name of that variable, you write a number. This number uniquely determines by which abstraction or product it is bound, in the following way. Look at all binders (abstractions and products) which have the variable in their scope (i.e. as subterm of M in $\Pi x{:}A.M$ or $\lambda x{:}A.M$), and count the number of these binders between the occurrence of the variable and the binder by which it is bound. We replace the variable by this number. Now the variables don't have names anymore, so we can write products and abstractions without specifying the bound variable, i.e. as $\Pi(A).M$ and $\lambda(A).M$. The best way to visualise this is using trees [de Bruijn, 1994]. The disadvantage of de Bruijn variables is a decreased readability of the term.

Example 6.1. We give some examples of pseudoterms written with named variables and de Bruijn variables (note that these terms are not well-typed).

$$\lambda x{:}*.\lambda y{:}*.yx \text{ becomes } \lambda(*).\lambda(*).01;$$
$$\Pi x{:}(\Pi y{:}*.y).(\lambda z{:}*.zx)x \text{ becomes } \Pi(\Pi(*).0).(\lambda(*).01)0;$$
$$\lambda x{:}*.x(\Pi z{:}(\Pi y{:}x.\lambda w{:}xy.xyw).xz)(\Pi w{:}x.xw) \text{ becomes } \lambda(*).0(\Pi(\Pi(0).\lambda(10).210).10)(\Pi(0).10);$$

Note that in the second line, the three zeroes all refer to different binders. In the third line the x in $\Pi y{:}x$ is not in the scope of the products, so the number 0 refers to the abstraction. $\varnothing$

For free variables one can just use a natural number which is too large to be bound within the term. A variable n which is in the scope of k binders is bound when $n < k$ and free when $n \geq k$. For a free variable we call $n - k$ its *index*. Two free variables are equal when they have the same index. This means that if we add enough binders in the front of the term, they will be bound by the same binder.

The main reason to use de Bruijn indices is that you don't have to deal with alpha conversion. If one uses named variables, one has to check for every substitution that the substitution is valid and one has to rename bound variables to fresh variables when the substitution is invalid. With de Bruijn variables there is no such problem, substitutions are always defined. Another advantage of de Bruijn indices is that there is a unique way to represent closed terms (a term is *closed* if it has no free variables.).

There's a small price one has to pay for using de Bruijn indices, and that is that one has to use a *lift operator* [Huet, 2011] (or shift operator [Abadi et al., 1991]). In the formalisation we denoted the lift operator by $M \uparrow n \# t$, where M is a term and n and t are natural numbers. Then $M \uparrow n \# t$ is the term M where n is added to all free variables with index at least t. This lift operator is necessary to define substitutions and for things like weakening.

One can use de Bruijn indices in a nice way for judgements. In judgements you don't have to name the variables in the context, so it is of the form

$$A_n, A_{n-1}, A_{n-2}, \ldots, A_0 \vdash_f M : N.$$

Here the variables in M and N with index 0 are bound by A_0, the variables with index 1 by A_1, and so on. So the weakening Lemma states that if we have the judgement above, then we also have the following judgement

$$A_n, \ldots, A_t, B, A_{t-1}, \ldots, A_0 \vdash_f M \uparrow 1 \# t : N \uparrow 1 \# t.$$

We need the lift of variables to ensure the variables are referring to the correct entry in the context.

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, lift operator and substitution	450
`f_env`	definition of contexts and substitution on contexts	250
`f_typ`	definition of the type system and the theory of Section 5.2	400
`f_typ2`	The theory of Sections 5.3–5.5	830
`f_equivalence`	the equivalence in Section 5.6 and product injectivity	300

In Appendix A one can find a summary of the Coq files, with most Definitions and the main Theorems.

7 Special cases of PTSs

In this chapter we will consider the specification **P** which was defined in Example 2.4. We repeat its definition here.

Definition 7.1. The specification **P** is defined in the following way. $\mathbf{P} = (\mathcal{S}_\mathbf{P}, \mathcal{A}_\mathbf{P}, \mathcal{R}_\mathbf{P})$ where

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

As stated in the introduction, this specification is of particular interest, because $\lambda\mathbf{P}$ is closely related to the logical framework LF [Harper et al., 1993]. It is also a member of the lambda cube.

7.1 Functional PTSs

An important property about $\lambda\mathbf{P}$ is that it is functional. We first prove some properties of all 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 $s_3 = s_3'$.

Every member of the lambda cube is functional, and most Pure Type Systems which are used in practice are functional.

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 means 'unique up to alpha conversion', which is as unique as you can get, since we identified terms up to alpha conversion.

Lemma 7.3 (Uniqueness of Types)**.** *If **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.

If M is a sort, then (M, N) and (M, N') are axioms. Since the PTS is functional. $N \equiv N'$.

If M is a variable, then $(M : N), (M : N') \in \Gamma$. Since variables occur once at most once in a context we conclude $N \equiv N'$.

If M is a product, say $M \equiv \Pi x{:}A.B$, then the last rule was (prod), so there are relations (s_1, s_2, s_3) and (s_1', s_2', s_3') such that

$$\begin{array}{lll} \Gamma \vdash_f A : s_1, & \Gamma, x : A \vdash_f B : s_2, & N = s_3 \\ \Gamma \vdash_f A : s_1', & \Gamma, x : A \vdash_f B : s_2', & N' = s_3' \end{array}$$

By the IH we know that $s_1 \equiv s_1'$ and $s_2 \equiv s_2'$ so $s_1 = s_1'$ and $s_2 = s_2'$. Since the PTS is functional we conclude that $N = s_3 = s_3' = N'$.

If M is an abstraction, say $M \equiv \lambda x{:}A.b$, then the last rule was (abs), so there are B, B' such that $\Gamma \vdash_f b : B$ and $\Gamma \vdash_f b : B'$ and $N \equiv \Pi x{:}A.B$ and $N' \equiv \Pi x{:}A.B'$. By the IH we know that $B \equiv B'$, hence $N \equiv N'$.

If M is an application, say $M \equiv Fa$, then the last rule was (app), hence $\Gamma \vdash F : \Pi x{:}A.B$ and $\Gamma \vdash F : \Pi x{:}A'.B'$ for some A, B, A', B' such that $N \equiv B[x := a]$ and $N' \equiv B'[x := a]$. By the IH $\Pi x{:}A.B \equiv \Pi x{:}A'.B'$, so $B \equiv B'$ hence $N \equiv N'$.

If M is a conversion, then $M \equiv a^H$ for some a, H. The last rule was (conv), hence $\Gamma \vdash_f H : A = N$ and $\Gamma \vdash_f H : A' = N'$ for some A, A'. By Lemma 5.8.1 we conclude that $N \equiv N'$. $\Box$

In functional specifications, if we have a convertibility proof between terms we also have a convertibility proof between the corresponding types

Proposition 7.4 (Equality between Types)**.** *If $\boldsymbol{S}$ is functional, $\Gamma \vdash_f M = M'$, $\Gamma \vdash_f M : T$, $\Gamma \vdash_f M' : T'$ then either $\Gamma \vdash_f T = T'$ or T and T' are equal 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.

If the last rule was (ref), then $M \equiv M'$, so by Uniqueness of Types we also have that $T \equiv T'$. Since T is a semitype, it is either a sort or has a type under Γ. In the first case we are done, and in the second case we can apply (ref).

If the last rule was (sym) or (trans) then the result follows by applying the same rule to all induction hypotheses.

If the last rule was (beta), then $M \equiv (\lambda x{:}A.b)a$ and $M' \equiv b[x := a]$. Let B be the type of b. Then both M and M' have type $B[x := a]$ and by Uniqueness of Types we know that $T \equiv B[x := a] \equiv T'$. We can apply (ref), since $B[x := a]$ has a type under Γ.

If the last rule was (prod-eq), then $M \equiv \Pi x{:}A.B$ and $M' \equiv \Pi x'{:}A'.B'$ concluded from the relations (s_1, s_2, s_3) and (s_1', s_2', s_3'), respectively. By the IH we know that $\Gamma \vdash s_1 = s_1'$ and $\Gamma \vdash s_2 = s_2'$. By Theorem 5.23 we know that these sorts are beta convertible, hence they must be equal: $s_1 = s_1'$ and $s_2 = s_2'$. Since the specification is functional, we know that also $s_3 = s_3'$, and these are the types of the products.

If the last rule was (abs-eq), then we can apply (prod-eq). The only hypothesis of (prod-eq) which is not trivial is the convertibility of B and $B'[x' := x^H]$, but this follows from the IH (in the case $B \equiv B'[x' := x^H]$ we can apply (ref), because these terms have a type under Γ).

If the last rule was (app-eq), then $M \equiv Fa$ and $M' \equiv F'a'$ where $\Gamma \vdash F : \Pi x{:}A.B$ and $\Gamma \vdash F' : \Pi x'{:}A'.B'$. By the IH, we know that these products are convertible, and hence by Product Injectivity (Corollary 5.24) we know that $\Gamma \vdash H : A = A'$ and $\Gamma, x : A \vdash B = B'[x' := x^H]$. By Substitution we conclude (note that $B'[x' := x^H][x := a] \equiv B'[x' := a^H]$) $\Gamma \vdash B[x := a] = B'[x' := a^H]$. We also know that $\Gamma \vdash a = a'$. Using (iota), (sym) and (trans) one obtains $\Gamma \vdash a^H = a'$. Now by Corollary 5.20 we conclude that $\Gamma \vdash B'[x' := a^H] = B'[x' := a']$. We finish this case by using (trans) and Uniqueness of Types.

If the last rule was (iota), then the statement is an assumption of the rule if we use Uniqueness of Types. □

Remark 7.5. Proposition 7.4 has some interesting consequences for functional type systems. Whenever one uses the rules (prod-eq) or (abs-eq) when $\mathbf{S}$ 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 $\lambda_f\mathbf{S}$.

$$\frac{\begin{array}{ll}\Gamma \vdash A : s_1 & \Gamma, x : A \vdash B : s_2 \\ \Gamma \vdash A' : s_1 & \Gamma, x' : A' \vdash B' : s_2 \\ \Gamma \vdash H : A = A' & \Gamma, x : A \vdash H' : B = B'[x' := x^H]\end{array}}{\Gamma \vdash \{H, [x : A]H'\} : \Pi x{:}A.B = \Pi x'{:}A'.B'}\ (s_1, s_2, s_3) \in \mathcal{R} \qquad \text{(prod-eq}'\text{)}$$

$$\frac{\begin{array}{ll}\Gamma \vdash_f A : s_1 & \Gamma, x : A \vdash_f b : B : s_2 \\ \Gamma \vdash_f A' : s_1 & \Gamma, x' : A' \vdash_f b' : B' : s_2 \\ \Gamma \vdash_f H : A = A' & \Gamma, x : A \vdash_f H' : b = b'[x' := x^H]\end{array}}{\Gamma \vdash_f \langle H, [x : A]H'\rangle : \lambda x{:}A.b = \lambda x'{:}A'.b'}\ (s_1, s_2, s_3) \in \mathcal{R} \qquad \text{(abs-eq}'\text{)}$$

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

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

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

1. Even though $\cdot \vdash_f a = b$, 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 when we would only allow (prod-eq) with $s_i = s'_i$. Because if we use (prod-eq′) twice one can prove

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

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

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 $\lambda_f\boldsymbol{P}$ 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 by Equality Typing (Lemma 2) we know that s has a type under Γ. Clearly $\Box$ cannot have a type in any context (because the last rule of the judgement must be (ax)), so $s = *$. 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.

If the last rule was (ref), then we're done. If the last rule was (sym) or (trans), then the result follows from the IH.

If the last rule was (beta), concluding $\Gamma \vdash_f H : (\lambda x{:}A.b)a = b[x := a]$, then $b[x := a] \equiv *$, which means that either $b \equiv *$ or $b \equiv x$ and $a \equiv *$. In either case, one of the hypotheses of the rule is $\Delta \vdash_f * : M : N$, which means that $\Box$, the only type of $*$, has a type, which is not possible.

The last rule cannot be (prod-eq), (abs-eq) or (app-eq), because these rules cannot equate $*$. If the last rule was (iota), then by an assumption there is an equality involving $\Box$, which is not possible.

2. Note that by Equality Typing, B has a type under Γ, say M. By Proposition 7.4 we know that either $M \equiv s$ or $\Gamma \vdash M = s$. In the second case we also have that $M \equiv s$, by Part 1 of this Proposition.

$\square$

Remark 7.8. Proposition 7.7 implies 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 7.9 demonstrates. $\varnothing$

Example 7.9. Given the specification $\mathbf{S} = (\mathcal{S}, \mathcal{A}, \mathcal{R})$ where

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

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

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

Also note that A^H does not have a second type (this follows from either Lemma 5.8.1 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 5.8.2) 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). $\varnothing$

8 Discussion

In this thesis we have shown that each PTS and is equivalent to its variant PTS$_f$. This means that given a PTS-term, we can reconstruct all conversion information between types 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 5.23 as a type-checking algorithm for the PTS. We did not give this algorithm explicitly, although since all our proofs are constructive, one should be able to make an algorithm using the proofs. It would be interesting to give this algorithm more explicitly.

Note that the equivalence is stronger than just saying that two convertible terms can be connected which a chain of one-step beta reductions between well-typed terms. This last statement is easily proven using Church-Rosser and Subject Reduction. The equivalence is stronger, because we also have to take into account that the terms are already annotated with convertibility proofs.

This thesis is an extension of [Geuvers and Wiedijk, 2008], where a system λH has been introduced. This system is a variant of $\lambda_f\mathbf{P}$, the PTS$_f$ corresponding to LF. The system λH also has terms that encode conversions, which are added to the proof terms in the same way as in PTS$_f$. In λH however, the terms in the equalities do not need to be well-typed in the sense of the system. An equality is of the form $H : A = A'$, where A and A' can be any pseudoterms, so they need not be well-typed. The mentioned paper left the equivalence between $\lambda_f\mathbf{P}$ and $\lambda\mathbf{P}$ as an open problem, but it proved the equivalence between λH and $\lambda\mathbf{P}$. However, we can generalise the system λH to arbitrary PTSs – in the same way we did with λF – getting a variant we can call PTS$_h$. We can then easily prove PTS$_f$ $\Rightarrow$ PTS$_h$ $\Rightarrow$ PTS, which establishes the equivalence of PTS$_h$ with the other systems.

As future research, it would be interesting to extend this work with other reductions than β-reduction. There are other reductions, like η-reduction, which is the compatible closure of the relation $\lambda x{:}A.fx \rightsquigarrow_\eta f$. We can also extend the system with *definitions* leading to δ-reduction or *inductive types* leading to ι-reduction. It is unknown to what extend the results in this work hold when we add these kinds of reductions. These reductions are of practical interest, because all practical implementations of type theory have definitions, and most of them also have inductive types.

For a PTS$_f$ we have come across an interesting problem whether the system with the rules (prod-eq) and (abs-eq) replaced by (prod-eq$'$) and (abs-eq$'$), as given in Remark 7.5, is equivalent to the original one. In Section 7.1 we have proven that this is the case for all functional specifications, but for non-functional specifications it is open.

9 Acknowledgments

Thanks to Egbert Rijke, Jaap van Oosten, James McKinna and Randy Pollack for partly reading preliminary versions of this work. Thanks to Vincent Siles for allowing to use his formalisation as the base for this formalisation. A big thanks to Herman Geuvers, for giving great advise on how to organise the presentation of this thesis, and to give a lot of comments after reading a preliminary version. I'm very grateful to Freek Wiedijk for his invaluable discussions, suggestions and ideas. Even though he says he didn't do a thing for this thesis (which couldn't be much further from the truth), he was a great supervisor.

Appendices

A Summary of the Coq formalisation

Here is a summary of the Coq formalisation. For the full version, see `http://www.cs.ru.nl/~freek/ptsf/`.

Due to technical reasons some notations in the formalisation are different that the corresponding notation in this thesis. $A \sim H$ means A^H, $M \cdot N$ is application of terms, $H_1 \cdot\mathrm{h}\, H_2$ is application of convertibility proofs, $H_1 \bullet H_2$ is the "composition" of convertibility proofs (i.e. via transitivity). Furthermore $\rho\, H$ is $\overline{H}$, the reflexivity proof and $\epsilon\, A$ is $|A|$, the erasure of A. $\#v$ denotes a variable and $!s$ a sort.

A.1 base

```
Parameter Sorts : Set.
Parameter Ax : Sorts → Sorts → Prop.
Parameter Rel : Sorts → Sorts → Sorts → Prop.
Definition Vars := nat.
```

A.2 f_term

```
Inductive Term : Set :=
  | Var : Vars → Term
  | Sort : Sorts → Term
  | Prod : Term → Term → Term
  | Abs : Term → Term → Term
  | App : Term → Term → Term
  | Conv : Term → Prf → Term
with Prf : Set :=
  | Refl : Term → Prf
  | Sym : Prf → Prf
  | Trans : Prf → Prf → Prf
  | Beta : Term → Prf
  | ProdEq : Prf → Term → Prf → Prf
  | AbsEq : Prf → Term → Prf → Prf
  | AppEq : Prf → Prf → Prf
  | Iota : Term → Prf.
Notation "! s" := (Sort s).
Notation "# v" := (Var v).
Notation "'Π' ( U ) , V " := (Prod U V).
Notation "'λ' [ U ] , v " := (Abs U v).
Notation "x · y" := (App x y).
Notation "A ∼ H" := (Conv A H).
Notation "'ρ' A" := (Refl A).
Notation "H †" := (Sym H).
Notation "H1 • H2" := (Trans H1 H2).
Notation "'β' A" := (Beta A).
Notation "{ H1 , [ A ] H2 }" := (ProdEq H1 A H2).
Notation "⟨ H1 , [ A ] H2 ⟩" := (AbsEq H1 A H2).
Notation "H1 ·h H2" := (AppEq H1 H2).
Notation "'ι' A" := (Iota A).

Fixpoint lift_rec (n:nat) (k:nat) (T:Term) {struct T}
:= match T with
  | # x ⇒ if le_gt_dec k x then Var (n+x) else Var x
  | ! s ⇒ Sort s
  | M · N ⇒ App (M ↑ n # k) (N ↑ n # k)
  | Π ( A ), B ⇒ Π (A ↑ n # k), (B ↑ n # (S k))
  | λ [ A ], M ⇒ λ [A ↑ n # k], (M ↑ n # (S k))
  | A ∼ H ⇒ A ↑ n # k ∼ H ↑h n # k
  end
  where "t ↑ n # k" := (lift_rec n k t)
with lift_rec_h (n:nat) (k:nat) (H:Prf) {struct H} :=
match H with
  | ρ A ⇒ ρ (A ↑ n # k)
  | H† ⇒ (H ↑h n # k)†
  | H•K ⇒ (H ↑h n # k)•(K ↑h n # k)
  | β A ⇒ β(A ↑ n # k)
  | H ·h K ⇒ (H ↑h n # k) ·h (K ↑h n # k)
  | {H,[A]K} ⇒ {H ↑h n # k,[A ↑ n # k]K ↑h n # (S
k)}
  | ⟨H,[A]K⟩ ⇒ ⟨H ↑h n # k,[A ↑ n # k]K ↑h n # (S
k)⟩
  | ι A ⇒ ι(A ↑ n # k)
  end
  where "t ↑h n # k" := (lift_rec_h n k t).
Notation " t ↑ n " := (lift_rec n 0 t).
Notation " t ↑h n " := (lift_rec_h n 0 t).

Fixpoint subst_rec U T n {struct T} :=
  match T with
  | # x ⇒ match (lt_eq_lt_dec x n) with
    | inleft (left _) ⇒ # x
    | inleft (right _) ⇒ U ↑ n
    | inright _ ⇒ # (x - 1)
  end
  | ! s ⇒ ! s
  | M · N ⇒ (M [ n ← U ]) · ( N [ n ← U ])
  | Π ( A ), B ⇒ Π ( A [ n ← U ] ), (B [ S n ← U
])
  | λ [ A ], M ⇒ λ [ A [ n ← U ] ], (M [ S n ← U
])
  | A ∼ H ⇒ A [ n ← U ] ∼ H [ n ←h U ]
end
  where " t [ n ← w ] " := (subst_rec w t n)
with subst_rec_h U H n {struct H} := match H with
  | ρ A ⇒ ρ A [ n ← U ]
  | H† ⇒ H [ n ←h U ] †
  | H•K ⇒ H[ n ←h U ]•K[ n ←h U ]
  | β A ⇒ β A[ n ← U ]
  | H ·h K ⇒ H[ n ←h U ] ·h K[ n ←h U ]
  | {H,[A]K} ⇒ {H[ n ←h U ],[A[ n ← U ]]K[ S
n ←h U ]}
  | ⟨H,[A]K⟩ ⇒ ⟨H[ n ←h U ],[A[ n ← U ]]K[ S n
←h U ]⟩
  | ι A ⇒ ι A [ n ← U ]
end
  where " t [ n ←h w ] " := (subst_rec_h w t n).
Notation " t [ ← w ] " := (subst_rec w t 0).
Notation " t [ ←h w ] " := (subst_rec_h w t 0).
```

A.3 f_env

```
Definition Env := list Term.
```

Inductive **item** (*A*:Type) (*x*:*A*): **list** *A* →**nat**→Prop :=
 | item_hd: ∀ Γ :**list** *A*, (**item** *x* (cons *x* Γ) O)
 | item_tl: ∀ (Γ:**list** *A*)(*n*:**nat**)(*y*:*A*), **item** *x* Γ *n* → **item** *x* (cons *y* Γ) (S *n*).
Notation " x ↓ n ∈ Γ " := (**item** *x* Γ *n*).

Inductive **trunc** (*A*:Type) : **nat**→**list** *A* →**list** *A*→Prop :=
 | trunc_O: ∀ (Γ:**list** *A*) , (**trunc** O Γ Γ)
 | trunc_S: ∀ (*k*:**nat**)(Γ Γ'*:**list** *A*)(*x*:*A*), **trunc** *k* Γ Γ'
 → **trunc** (S *k*) (cons *x* Γ) Γ'.

Inductive **ins_in_env** (Γ:Env) (*d1*:**Term**): **nat**→Env → Env →Prop :=
 | ins_O: **ins_in_env** Γ *d1* O Γ (*d1*::Γ)
 | ins_S: ∀ (*n*:**nat**)(Δ Δ':Env)(*d*:**Term**), (**ins_in_env** Γ *d1* *n* Δ Δ')
 → **ins_in_env** Γ *d1* (S *n*) (*d*::Δ) ((*d* ↑ 1 # *n*)::Δ').

Definition item_lift (*t*:**Term**) (Γ:Env) (*n*:**nat**) :=
 ∃ *u* , *t*= *u* ↑ (S *n*) ∧ *u* ↓ *n* ∈ Γ .
Notation " t ↓ n ⊂ Γ " := (item_lift *t* Γ *n*).

Inductive **sub_in_env** (Γ : Env) (*t* *T*:**Term**):
 nat → Env → Env → Prop :=
 | sub_O : **sub_in_env** Γ *t* *T* 0 (*T* :: Γ) Γ
 | sub_S :
 ∀ Δ Δ' *n* *B*,
 sub_in_env Γ *t* *T* *n* Δ Δ' →
 sub_in_env Γ *t* *T* (S *n*) (*B* :: Δ) (*B* [*n*← *t*] :: Δ').

A.4 f_typ

Inductive wf : *Env* → Prop :=
 | *wf_nil* : *nil* ⊢
 | *wf_cons* : ∀ Γ *A s*, Γ ⊢ *A* : !*s* → *A*::Γ ⊢
where "Γ ⊢" := (wf Γ)
with *typ* : *Env* → *Term* → *Term* → Prop :=
 | *cSort* : ∀ Γ *s t*, *Ax s t* → Γ ⊢ → Γ ⊢ !*s* : !*t*
 | *cVar* : ∀ Γ *A v*, Γ ⊢ → *A* ↓ *v* ⊂ Γ → Γ ⊢ #*v* : *A*
 | *cProd* : ∀ Γ *A B s1 s2 s3*, *Rel s1 s2 s3* → Γ ⊢ *A* : !*s1* → *A*::Γ ⊢ *B* : !*s2* → Γ ⊢ Π(*A*), *B* : !*s3*
 | *cAbs* : ∀ Γ *A B b s1 s2 s3*, *Rel s1 s2 s3* → Γ ⊢ *A* : !*s1* → *A*::Γ ⊢ *b* : *B* → *A*::Γ ⊢ *B* : !*s2* → Γ ⊢ λ[*A*], *b* : Π(*A*), *B*
 | *cApp* : ∀ Γ *F a A B* , Γ ⊢ *F* : Π(*A*), *B* → Γ ⊢ *a* : *A* → Γ ⊢ *F* · *a* : *B*[←*a*]
 | *cConv* : ∀ Γ *a A B s H*, Γ ⊢ *a* : *A* → Γ ⊢ *B* : !*s* → Γ ⊢ *H* : *A* = *B* → Γ ⊢ *a* ∼ *H* : *B*
where "Γ ⊢ t : T" := (*typ* Γ *t T*)
with *typ_h* : *Env* → *Prf* → *Term* → *Term* → Prop :=
 | *cRefl* : ∀ Γ *a A*, Γ ⊢ *a* : *A* → Γ ⊢ ρ *a* : *a* = *a*
 | *cSym* : ∀ Γ *H A B*, Γ ⊢ *H* : *A* = *B* → Γ ⊢ *H*† : *B* = *A*
 | *cTrans* : ∀ Γ *H K A B C*, Γ ⊢ *H* : *A* = *B* → Γ ⊢ *K* : *B* = *C* → Γ ⊢ *H*•*K* : *A* = *C*
 | *cBeta* : ∀ Γ *a A b B s1 s2 s3*, *Rel s1 s2 s3* → Γ ⊢ *a* : *A* → Γ ⊢ *A* : !*s1*
 → *A*::Γ ⊢ *b* : *B* → *A*::Γ ⊢ *B* : !*s2* → Γ ⊢ β((λ[*A*], *b*)·*a*) : (λ[*A*], *b*)·*a* = *b*[←*a*]
 | *cProdEq* : ∀ Γ *A A' B B' H K s1 s2 s3 s1' s2' s3'*, *Rel s1 s2 s3* → *Rel s1' s2' s3'*
 → Γ ⊢ *A* : !*s1* → Γ ⊢ *A'* : !*s1'* → *A*::Γ ⊢ *B* : !*s2* → *A'*::Γ ⊢ *B'* : !*s2'*
 → Γ ⊢ *H* : *A* = *A'* → *A*::Γ ⊢ *K* : *B* = (*B'*↑1#1)[←#0∼*H*↑*h1*] → Γ ⊢ {*H*,[*A*]*K*} : Π(*A*), *B* = Π(*A'*), *B'*
 | *cAbsEq* : ∀ Γ *A A' b b' B B' H K s1 s2 s3 s1' s2' s3'*, *Rel s1 s2 s3* → *Rel s1' s2' s3'*
 → Γ ⊢ *A* : !*s1* → Γ ⊢ *A'* : !*s1'* → *A*::Γ ⊢ *b* : *B* → *A'*::Γ ⊢ *b'* : *B'* → *A*::Γ ⊢ *B* : !*s2* → *A'*::Γ ⊢ *B'* : !*s2'*
 → Γ ⊢ *H* : *A* = *A'* → *A*::Γ ⊢ *K* : *b* = (*b'*↑1#1)[←#0∼*H*↑*h1*] → Γ ⊢ ⟨*H*,[*A*]*K*⟩ : λ[*A*], *b* = λ[*A'*], *b'*
 | *cAppEq* : ∀ Γ *F F' a a' A A' B B' H K*, Γ ⊢ *F* : Π(*A*), *B* → Γ ⊢ *F'* : Π(*A'*), *B'* → Γ ⊢ *a* : *A* → Γ ⊢ *a'* : *A'*
 → Γ ⊢ *H* : *F* = *F'* → Γ ⊢ *K* : *a* = *a'* → Γ ⊢ *H* ·*h* *K* : *F* · *a* = *F'* · *a'*
 | *cIota* : ∀ Γ *a A B s H*, Γ ⊢ *a* : *A* → Γ ⊢ *B* : !*s* → Γ ⊢ *H* : *A* = *B* → Γ ⊢ ι(*a*∼*H*) : *a* = *a*∼*H*
where "Γ ⊢ H : A = B" := (*typ_h* Γ *H A B*).

Lemma *wf_typ* : ∀ Γ *t T*, Γ ⊢ *t* : *T* → Γ ⊢.

Theorem *weakening*:
(∀ Γ *M N*, Γ ⊢ *M* : *N* → ∀ Δ *A s n* Γ', *ins_in_env* Δ *A n* Γ Γ' → Δ ⊢ *A* : !*s* → Γ' ⊢ *M* ↑ 1 # *n* : *N* ↑ 1 # *n*) ∧
(∀ Γ *H M N*, Γ ⊢ *H* : *M* = *N* → ∀ Δ *A s n* Γ', *ins_in_env* Δ *A n* Γ Γ' → Δ ⊢ *A* : !*s* → Γ' ⊢ *H* ↑*h* 1 # *n* : *M* ↑ 1 # *n* = *N* ↑ 1 # *n*) ∧
(∀ Γ , Γ ⊢ → ∀ Δ *A s n* Γ', *ins_in_env* Δ *A n* Γ Γ' → Δ ⊢ *A* : !*s* → Γ' ⊢).

Theorem *thinning_n* : ∀ *n* Δ Δ', *trunc n* Δ Δ' → ∀ *M T* , Δ' ⊢ *M* : *T* → Δ ⊢ → Δ ⊢ *M* ↑ *n* : *T* ↑ *n*.

Theorem *substitution* :
(∀ Γ *M N* , Γ ⊢ *M* : *N* → ∀ Δ *a A* Γ' *n*, Δ ⊢ *a* : *A* → *sub_in_env* Δ *a A n* Γ Γ' → Γ ⊢ → Γ' ⊢ *M* [*n* ← *a*] : *N* [*n* ← *a*]) ∧
(∀ Γ *H M N* , Γ ⊢*H*:*M* = *N* → ∀ Δ *a A* Γ' *n*, Δ ⊢ *a* : *A* → *sub_in_env* Δ *a A n* Γ Γ' → Γ ⊢ → Γ' ⊢ *H*[*n*←*h* *a*]:*M* [*n* ← *a*] = *N* [*n* ← *a*]) ∧
(∀ Γ , Γ ⊢ → ∀ Δ *a A* Γ' *n*, Δ ⊢ *a* : *A* → *sub_in_env* Δ *a A n* Γ Γ' → Γ' ⊢).

Lemma *wf_item* : ∀ Γ *A n*, *A* ↓ *n* ∈ Γ → ∀ Γ', Γ ⊢ → *trunc* (*S n*) Γ Γ' → ∃ *s*, Γ' ⊢ *A* : !*s*.

Lemma *equality_unique* : ∀ Γ *H A B C D*, Γ ⊢ *H* : *A* = *B* → Γ ⊢ *H* : *C* = *D* → *A* = *C* ∧ *B* = *D*.
Definition *has_type A* Γ := (∃ *B*, Γ ⊢ *A* : *B*).
Lemma *equality_typing* : ∀ Γ *H A B*, Γ ⊢ *H* : *A* = *B* → *has_type A* Γ ∧ *has_type B* Γ.

Definition *semitype A* Γ := (∃ *s*,*A*=!*s*)∨(∃ *s*, Γ ⊢ *A* : !*s*).
Theorem *TypeCorrect* : ∀ Γ *M N*, Γ ⊢ *M* : *N* → *semitype N* Γ.

A.5 f_typ2

A.5.1 Erasure

Below, UTM.Term and UEM.Env are terms/contexts without convertiblity proofs, while TM.Term and EM.Env are terms/contexts with convertibility proofs. The Lemma context_conversion is not stated explicitely in my thesis because it is not required for the proof of the equivalence, but when I had already formalised it, it was convenient to use it for the Theorem PTSeq2PTSF.

Lemma subst_typ : ∀ Δ *A1 A2 n* Γ Γ*1* Γ*2 M N H s*, Γ

⊢ M:N→Δ ⊢ H : A2=A1→Δ ⊢ A2:!s
→**ins_in_env** Δ A2 (S n) Γ Γ1→**sub_in_env** (A2::Δ) (#0 ∼ H ↑h 1) (A1↑1) n Γ1 Γ2→
Γ2 ⊢ (M ↑ 1 # (S n)) [n ← #0 ∼ H ↑h 1] : (N ↑ 1 # (S n)) [n ← #0 ∼ H ↑h 1].

Lemma subst_wf : ∀ Δ A1 A2 n Γ Γ1 Γ2 H s, Γ ⊢→Δ ⊢ H : A2=A1→Δ ⊢ A2:!s
→**ins_in_env** Δ A2 (S n) Γ Γ1→**sub_in_env** (A2::Δ) (#0 ∼ H ↑h 1) (A1↑1) n Γ1 Γ2→
Γ2 ⊢.

Lemma subst_eq : ∀ Δ A1 A2 n Γ Γ1 Γ2 H2 M N H s, Γ ⊢ H2 : M=N→Δ ⊢ H : A2=A1→Δ ⊢ A2:!s
→**ins_in_env** Δ A2 (S n) Γ Γ1→**sub_in_env** (A2::Δ) (#0 ∼ H ↑h 1) (A1↑1) n Γ1 Γ2→
Γ2 ⊢ (H2 ↑h 1 # (S n)) [n ←h #0 ∼ H ↑h 1] : (M ↑ 1 # (S n)) [n ← #0 ∼ H ↑h 1] = (N ↑ 1 # (S n)) [n ← #0 ∼ H ↑h 1].

Fixpoint erasure (T:**TM.Term**) {struct T} : **UTM.Term** := match T with
| # x ⇒ (# x)%UT
| ! s ⇒ (! s)%UT
| M · N ⇒ ((ϵ M) · (ϵ N))%UT
| Π (A), B ⇒ (Π (ϵ A), (ϵ B))%UT
| λ [A], M ⇒ (λ [ϵ A], (ϵ M))%UT
| A ∼ H ⇒ ϵ A
end
where "'ϵ' t" := (erasure t).
Fixpoint erasure_context (Γ:EM.Env) {struct Γ} : UEM.Env := match Γ with
| nil ⇒ nil
| A::Γ ⇒ ϵ A::ϵc Γ
end
where "'ϵc' t" := (erasure_context t).

Theorem PTSF2PTS : (∀ Γ A B,Γ ⊢ A : B → (ϵc Γ ⊢ ϵ A : ϵ B)%UT)∧
(∀ Γ H A B, Γ ⊢ H : A = B → ϵ A ≡ ϵ B)∧
(∀ Γ, Γ ⊢→ ϵc Γ ⊢ %UT).

Proposition erasure_injectivity_term : ∀ a b Γ A B,Γ ⊢ a : A→Γ ⊢ b : B→ϵ a=ϵ b→∃ H , Γ ⊢ H : a = b.

Lemma erasure_injectivity_term_sort : ∀ A Γ B s,Γ ⊢ A : B→ϵ A=!s%UT→Γ ⊢ A = !s.

Lemma erasure_term : ∀ A1 A2 Γ a1,Γ ⊢ a1 : A1→ϵ A1=ϵ A2→semitype A2 Γ→∃ a2,ϵ a2=ϵ a1∧Γ ⊢ a2 : A2.

Lemma erasure_term_type : ∀ Γ a1 A1 A2 B s,Γ ⊢ a1 : A1→Γ ⊢ A2 : B→ϵ A1=ϵ A2→ϵ B=!s%UT
→∃ A a,ϵ A=ϵ A1∧ϵ a=ϵ a1∧Γ ⊢ a : A : !s.

Lemma erasure_equality : ∀ Γ a1 a2 A B H,Γ ⊢ H : a1 = a2→Γ ⊢ a1 : A→Γ ⊢ a2 : A→ϵ A=ϵ B→semitype B Γ
→∃ b1 b2,ϵ b1=ϵ a1∧ϵ b2=ϵ a2∧Γ ⊢ b1 : B∧Γ ⊢ b2 : B∧Γ ⊢ b1 = b2.

Lemma context_conversion :
(∀ Γ A B,Γ ⊢ A : B→∀ Δ,Δ ⊢→ϵc Γ = ϵc Δ→∃ A' B',ϵ A'=ϵ A∧ϵ B'=ϵ B∧Δ ⊢ A' : B')∧
(∀ Γ H A B,Γ ⊢ H : A = B→∀ Δ,Δ ⊢→ϵc Γ = ϵc Δ→∃ H' A' B',ϵ A'=ϵ A∧ϵ B'=ϵ B∧Δ ⊢ H' : A' = B')∧
(∀ Γ, Γ ⊢ →**True**).

A.5.2 Multiple substitutions

We define multiple substitutions for contexts, terms and convertibility proofs. Note that for contexts we require the list of convertibility to make sense, but we do not do this for terms and convertibility proofs. Then we prove Lemmas as in Section 5.5.

Inductive **subst_mult_env** : **nat**→Env→**list Prf**→Env→Prop :=
| msub_O : ∀ n Γ,**subst_mult_env** n Γ nil Γ
| msub_S : ∀ n Γ H HH Γ' Γs Γ1 Γ2 A1 A2 s, **subst_mult_env** (S n) Γ' HH Γ →**trunc** (S n) Γ Γs→ Γs ⊢ A2 : !s → Γs ⊢ H : A2 = A1 → **ins_in_env** Γs A2 (S n) Γ Γ1→
sub_in_env (A2::Γs) (#0 ∼ H ↑h 1) (A1↑1) n Γ1 Γ2→ **subst_mult_env** n Γ' (H::HH) Γ2.

Fixpoint subst_mult_term (n:**nat**) (M:**Term**) (HH:**list Prf**) {struct HH} := match HH with
| nil ⇒ M
| H'::HH' ⇒ (subst_mult_term (S n) M HH') ↑ 1 # (S n) [n ← #0 ∼ H' ↑h 1]
end.

Fixpoint subst_mult_prf (n:**nat**) (H:**Prf**) (HH:**list Prf**) {struct HH} := match HH with
| nil ⇒ H
| H'::HH' ⇒ (subst_mult_prf (S n) H HH') ↑h 1 # (S n) [n ←h #0 ∼ H' ↑h 1]
end.

Lemma subst_mult_typ : ∀ n Γ HH Γ' M' N', Γ' ⊢ M' : N' → **subst_mult_env** n Γ' HH Γ →
Γ ⊢ subst_mult_term n M' HH : subst_mult_term n N' HH.
Lemma subst_mult_eq : ∀ n Γ HH Γ' H' M' N', Γ' ⊢ H' : M' = N' → **subst_mult_env** n Γ' HH Γ
→ Γ ⊢ subst_mult_prf n H' HH : subst_mult_term n M' HH = subst_mult_term n N' HH.

Lemma equality_subst_ext : ∀ Δ Γ' M N a1 a2 T K,Δ ⊢ a1 : T → Δ ⊢ a2 : T → Δ ⊢ K : a1 = a2 → Γ' ⊢ M : N →
∀ Γ1 Γ2 HH, **sub_in_env** Δ a1 T (length HH) Γ' Γ1→**sub_in_env** Δ a2 T (length HH) Γ' Γ2
→**subst_mult_env** 0 Γ2 HH Γ1→ Γ1 ⊢ (M [length HH ← a1]) = subst_mult_term 0 M[length HH ← a2] HH.

Corollary equality_subst : ∀ Γ F N H M1 M2 A,A::Γ ⊢ F : N→Γ ⊢ H : M1 = M2→Γ ⊢ M1 : A→Γ ⊢ M2 : A → Γ ⊢ F [← M1] = F [← M2].

A.5.3 Unique derivation

We define the term comparable. Then we define derivation trees. In the formalisation a derivation tree **deriv** is a finite tree labelled with natural numbers, with an arbitrary number of branches at each node. Then we define how a derivation tree corresponds to a judgement.

der_wf D Γ states that D is a derivation tree of the judgement $\Gamma \vdash_f$.

der_typ D Γ M N states that D is a derivation tree of the judgement $\Gamma \vdash_f M : N$.

der_h D Γ H M N states that D is a derivation tree of the judgement $\Gamma \vdash_f H : M = N$.

Inductive **comparable** : **Term** → **Term** → Prop :=
| comp_refl : ∀ M, **comparable** M M
| comp_sort : ∀ s t, **comparable** !s !t
| comp_prod : ∀ A M N, **comparable** M N → **comparable**

(Π(*A*),*M*) (Π(*A*),*N*).

Lemma type_comparable : ∀ Γ *M A*, Γ ⊢ *M* : *A* → ∀ *B*, Γ ⊢ *M* : *B* → **comparable** *A B*.

Inductive **deriv** : Set :=
| node : **list deriv** → **nat** → **deriv**.
Inductive **der_wf** : **deriv** → Env → Prop :=
| dnil : **der_wf** (node nil 0) nil
| dcons : ∀ Γ *A s D*, **der_typ** *D* Γ *A* !*s* → **der_wf** (node ([*D*]) 1) (*A*::Γ)
with **der_typ** : **deriv** → Env → **Term** → **Term** → Prop :=
| dSort : ∀ Γ *s t D*, *Ax s t* → **der_wf** *D* Γ → **der_typ** (node ([*D*]) 2) Γ !*s* !*t*
| dVar : ∀ Γ *A v D*, *A* ↓ *v* ⊂ Γ → **der_wf** *D* Γ → **der_typ** (node ([*D*]) 3) Γ #*v A*
| dProd : ∀ Γ *A B s1 s2 s3 D1 D2*, *Rel s1 s2 s3* → **der_typ** *D1* Γ *A* !*s1* → **der_typ** *D2* (*A*::Γ) *B* !*s2* → **der_typ** (node ([*D1*;*D2*]) 4) Γ (Π(*A*), *B*) !*s3*
| dAbs : ∀ Γ *A B b s1 s2 s3 D1 D2 D3*, *Rel s1 s2 s3* → **der_typ** *D1* Γ *A* !*s1* → **der_typ** *D2* (*A*::Γ) *b B*
→ **der_typ** *D3* (*A*::Γ) *B* !*s2* → **der_typ** (node ([*D1*;*D2*;*D3*]) 5) Γ (λ[*A*], *b*) (Π(*A*), *B*)
| dApp : ∀ Γ *F a A B D1 D2*, **der_typ** *D1* Γ *F* (Π(*A*), *B*) → **der_typ** *D2* Γ *a A* → **der_typ** (node ([*D1*;*D2*]) 6) Γ (*F* · *a*) (*B*[←*a*])
| dConv : ∀ Γ *a A B s H D1 D2 D3*, **der_typ** *D1* Γ *a A* → **der_typ** *D2* Γ *B* !*s* → **der_h** *D3* Γ *H A B* → **der_typ** (node ([*D1*;*D2*;*D3*]) 7) Γ (*a* ∼ *H*) *B*
with **der_h** : **deriv** → Env → **Prf** → **Term** → **Term** → Prop :=
| dRefl : ∀ Γ *a A D*, **der_typ** *D* Γ *a A* → **der_h** (node ([*D*]) 8) Γ (ρ *a*) *a a*
| dSym : ∀ Γ *H A B D*, **der_h** *D* Γ *H A B* → **der_h** (node ([*D*]) 9) Γ (*H*†) *B A*
| dTrans : ∀ Γ *H K A B C D1 D2*, **der_h** *D1* Γ *H A B* → **der_h** *D2* Γ *K B C* → **der_h** (node ([*D1*;*D2*]) 10) Γ (*H*•*K*) *A C*
| dBeta : ∀ Γ *a A b B s1 s2 s3 D1 D2 D3 D4*, *Rel s1 s2 s3* → **der_typ** *D1* Γ *a A* → **der_typ** *D2* Γ *A* !*s1*
→ **der_typ** *D3* (*A*::Γ) *b B* → **der_typ** *D4* (*A*::Γ) *B* !*s2* → **der_h** (node ([*D1*;*D2*;*D3*;*D4*]) 11) Γ (β((λ[*A*], *b*)·*a*)) ((λ[*A*], *b*)·*a*) (*b*[←*a*])
| dProdEq : ∀ Γ *A A' B B' H K s1 s2 s3 s1' s2' s3' D1 D2 D3 D4 D5 D6*, *Rel s1 s2 s3* → *Rel s1' s2' s3'*
→ **der_typ** *D1* Γ *A* !*s1* → **der_typ** *D2* Γ *A'* !*s1'* → **der_typ** *D3* (*A*::Γ) *B* !*s2* → **der_typ** *D4* (*A'*::Γ) *B'* !*s2'*
→ **der_h** *D5* Γ *H A A'* → **der_h** *D6* (*A*::Γ) *K B* ((*B'*↑1#1)[←#0∼*H*↑h1])
→ **der_h** (node ([*D1*;*D2*;*D3*;*D4*;*D5*;*D6*]) 12) Γ ({*H*,[*A*]*K*}) (Π(*A*), *B*) (Π(*A'*), *B'*)
| dAbsEq : ∀ Γ *A A' b b' B B' H K s1 s2 s3 s1' s2' s3' D1 D2 D3 D4 D5 D6 D7 D8*, *Rel s1 s2 s3* → *Rel s1' s2' s3'*
→ **der_typ** *D1* Γ *A* !*s1* → **der_typ** *D2* Γ *A'* !*s1'* → **der_typ** *D3* (*A*::Γ) *b B* → **der_typ** *D4* (*A'*::Γ) *b' B'*
→ **der_typ** *D5* (*A*::Γ) *B* !*s2* → **der_typ** *D6* (*A'*::Γ) *B'* !*s2'* → **der_h** *D7* Γ *H A A'* → **der_h** *D8* (*A*::Γ) *K b* ((*b'*↑1#1)[←#0∼*H*↑h1])
→ **der_h** (node ([*D1*;*D2*;*D3*;*D4*;*D5*;*D6*;*D7*;*D8*]) 13) Γ (⟨*H*,[*A*]*K*⟩) (λ[*A*], *b*) (λ[*A'*], *b'*)
| dAppEq : ∀ Γ *F F' a a' A A' B B' H K D1 D2 D3 D4 D5 D6*, **der_typ** *D1* Γ *F* (Π(*A*), *B*) → **der_typ** *D2* Γ *F'* (Π(*A'*), *B'*)
→ **der_typ** *D3* Γ *a A* → **der_typ** *D4* Γ *a' A'* → **der_h** *D5* Γ *H F F'* → **der_h** *D6* Γ *K a a'*
→ **der_h** (node ([*D1*;*D2*;*D3*;*D4*;*D5*;*D6*]) 14) Γ (*H* ·h *K*) (*F* · *a*) (*F'* · *a'*)
| dIota : ∀ Γ *a A B s H D1 D2 D3*, **der_typ** *D1* Γ *a A* → **der_typ** *D2* Γ *B* !*s* → **der_h** *D3* Γ *H A B* → **der_h** (node ([*D1*;*D2*;*D3*]) 15) Γ (ι(*a*∼*H*)) *a* (*a*∼*H*).

Theorem unique_der_ext : (∀ *D* Γ *A B*, **der_typ** *D* Γ *A B* → ∀ *A0 B0 D0*, **der_typ** *D0* Γ *A0 B0* → **comparable** *A A0* → *D* = *D0*)∧
(∀ *D* Γ *H A B*, **der_h** *D* Γ *H A B* → ∀ *A0 B0 D0*, **der_h** *D0* Γ *H A0 B0* → *D* = *D0*)∧
(∀ *D* Γ , **der_wf** *D* Γ → ∀ *D0*, **der_wf** *D0* Γ → *D* = *D0*).

Theorem unique_der : (∀ *D D0* Γ *A B*, **der_typ** *D* Γ *A B* → **der_typ** *D0* Γ *A B* → *D* = *D0*)∧
(∀ *D D0* Γ *H A B*, **der_h** *D* Γ *H A B* → **der_h** *D0* Γ *H A B* → *D* = *D0*)∧
(∀ *D D0* Γ , **der_wf** *D* Γ → **der_wf** *D0* Γ → *D* = *D0*).

A.6 f_equivalence

Here we use some notation and definitions of Siles' formalisation, which I did not include in this appendix. The most notations should be obvious, the suffix %UT means it is the typing without convertibility proofs. The turnstile with a prime means the judgements generated by the rules in Figure 2.2. Theorem PTSeq2PTSF is formulated differently than Theorem 5.21 because in the formalisation we already proved the Lemma context_conversion.

Theorem PTSeq2PTSF : (∀ Γ *M N* , Γ ⊢e *M* : *N* →∃ Γ' *M' N'* ,ϵc Γ'=Γ∧ϵ *M'*=*M*∧ϵ *N'*=*N*∧ Γ' ⊢ *M'* : *N'*)∧
(∀ Γ *M N A*, Γ ⊢e *M* = *N* : *A*→∃ Γ' *H M' N' A'*,ϵc Γ'=Γ∧ϵ *M'*=*M*∧ϵ *N'*=*N*∧ϵ *A'*=*A*∧Γ' ⊢ *M'* : *A'*∧Γ' ⊢ *N'* : *A'*∧Γ' ⊢ *H* : *M'* = *N'*)∧
(∀ Γ , Γ ⊢e →∃ Γ' ,ϵc Γ'=Γ∧ Γ' ⊢).

Theorem PTSl2PTSF : (∀ Γ *M N*,(Γ ⊢' *M* : *N*)%*UT* → ∃ Γ' *M' N'*, ϵc Γ'=Γ∧ϵ *M'*=*M*∧ϵ *N'*=*N*∧Γ' ⊢ *M'* : *N'*)∧
(∀ Γ *M N*,(∃ *A B*,(Γ ⊢' *M* : *A*)%*UT*∧(Γ ⊢' *N* : *B*)%*UT*∧ *M* ≡ *N*)→ ∃ Γ' *M' N'*, ϵc Γ'=Γ∧ϵ *M'*=*M*∧ϵ *N'*=*N*∧Γ' ⊢ *M'* = *N'*)∧
(∀ Γ ,(∃ *M N*,(Γ ⊢' *M* : *N*)%*UT*) → ∃ Γ' , ϵc Γ'=Γ∧ Γ' ⊢).

Theorem PTSlequivPTSF : (∀ Γ *M N*,(Γ ⊢' *M* : *N*)%*UT* ↔ ∃ Γ' *M' N'*, ϵc Γ'=Γ∧ϵ *M'*=*M*∧ϵ *N'*=*N*∧Γ' ⊢ *M'* : *N'*)∧
(∀ Γ *M N*,(∃ *A B*,(Γ ⊢' *M* : *A*)%*UT*∧(Γ ⊢' *N* : *B*)%*UT*∧ *M* ≡ *N*)<-> ∃ Γ' *M' N'*, ϵc Γ'=Γ∧ϵ *M'*=*M*∧ϵ *N'*=*N*∧Γ' ⊢ *M'* = *N'*)∧
(∀ Γ ,(Γ ⊢')%*UT* ↔ ∃ Γ' , ϵc Γ'=Γ∧ Γ' ⊢).

Theorem Prod_Injective : ∀ Γ *A B A' B' H*, Γ ⊢ *H* : Π(*A*), *B* = Π(*A'*), *B'* → ∃ *H K*, Γ ⊢ *H* : *A* = *A'* ∧ *A*::Γ ⊢ *K* : *B* = (*B'*↑1#1)[←#0∼*H*↑h1].

Bibliography

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, March 2006.

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

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

Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In *FM 2006: Formal Methods*, pages 460–475. Springer, 2006.

Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2): 56–68, 1940.

Thierry Coquand and Gérard Huet. The calculus of constructions. Technical Report 530, INRIA, 1986.

Nicolaas Govert 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.

Nicolaas Govert de Bruijn. Generalizing automath by means of a lambda-typed lambda calculus. In *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 313 – 337. Elsevier, 1994.

Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

Herman Geuvers and Freek Wiedijk. A logical framework with explicit conversions. *Electronic Notes in Theoretical Computer Science*, 199:33 – 47, 2008.

Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, 1972.

Georges Gonthier. A computer-checked proof of the four colour theorem. *preprint*, 2005.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40 (1):143–184, January 1993.

William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

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

Per Martin-Löf. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, Italy, 1984.

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

Conor 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.

Henri Poincaré. *Science and hypothesis*. Science Press, 1905.

John C Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425, 1974.

Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(2):153 – 180, May 2012.

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