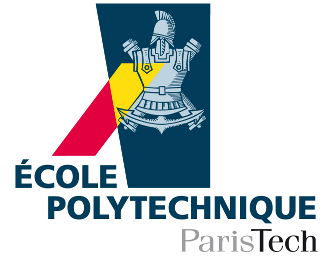

École Polytechnique
Thèse de Doctorat
Spécialité Informatique

Investigation on the typing of equality in type systems

Présentée et soutenue publiquement par

Vincent Siles

le 25 Novembre 2010

devant le jury composé de

Bruno	Barras	Co-directeur de thèse
Gilles	Barthe	Rapporteur
Roberto	Di Cosmo	
Herman	Geuvers	Rapporteur
Hugo	Herbelin	Co-directeur de thèse
Alexandre	Miquel	Rapporteur
Randy	Pollack	
Benjamin	Werner	

Abstract

Pure Type Systems are a good way to factorize the questions of meta-theory about a large family of type systems. They have been introduced as a generalization of Barendregt's λ-cube, an abstraction of several type systems like the *Simply Typed λ-Calculus*, *System F* or the *Calculus of Constructions*. One critical detail of the Pure Type Systems is their *conversion rule* that allows to do computation at the level of types.

Traditionally, Pure Type Systems are presented in a natural deduction style, and use an *untyped* notion of conversion. Through the years, several presentations of the Pure Type Systems have been used, with subtle differences like sequent calculus instead of natural deduction, or the use of a *typed* conversion instead of the untyped original one. The question to know whereas the latter choice leads to equivalent systems has been first asked by Geuvers in the early 90's, and the answer was only known for particular subclasses of Pure Type Systems. The main contribution of this dissertation is to finally provide a final and positive answer to this question by proving that all Pure Type Systems relying on an untyped conversion are equivalent to their typed conversion counterpart. During this work, we also investigated the open problem of *Expansion Postponement*.

The proofs are quite complex and rely on mechanisms that are tedious to manually check. In order to have more confidence in our development, all the results presented in this dissertation have also been checked within the proof assistant *Coq*.

Acknowledgments

Ben Denis Marie Cecile Pierre Nico Arnaud Capcom Claire Popa Ps3 Danko Elie Bruno Hugo Lisa Mathieu Moman Belette Krabou Frerot Marthe Nimmy Guillaume Jean-Jacques theblatte Typhaine Py Stephane kaisse Typical Mathias Marie-Loutre TheDoctor Thomas James pi.r2 Jury Marion julio Andreas sos David Barbara tchii Soeur tonfa Yann Aude

Contents

I Introduction 1

1 Type systems in a nutshell 9
1.1 The general setting 10
1.2 The purpose of programming languages 11
1.3 A practical example 13
1.4 Computation on types 18
1.5 Where to go next ? 20

II Untyped equality 21

2 Pure Type Systems 23
2.1 Pure Type System in Natural Deduction 25
2.1.1 Terms and Untyped Reductions 25
2.1.2 Presentation of Pure Type Systems 28
2.2 Pure Type Systems in Sequent Calculus 36
2.2.1 Terms and Reduction 38
2.2.2 Confluence of β-reduction in $\overline{\lambda}$ 39
2.2.3 Typing Rules 42
2.2.4 Properties of the system 42
2.3 Delayed Pure Type System in Sequent Calculus 45
2.3.1 Typing Rules 46
2.3.2 Properties of the system 46
2.4 Expansion Postponement in Delayed System 50
2.4.1 Postponement in Sequent Calculus 51
2.4.2 Postponement in Natural Deduction 52
2.5 Sequent Calculus and Type Inference 54
2.6 A brief look back at syntactical Pure Type Systems 57

III Typed equality 59

3 Judgmental Equality 61

3.1 PTSs with Judgmental Equality 63
3.1.1 Typing Rules 63
3.1.2 Subject Reduction and Equivalence 65
3.2 Basic meta-theory of the annotated system 68
3.2.1 Definition of PTSs with Annotated Type Reduction . . 69
3.2.2 General properties of PTS_{atr} 74
3.2.3 The *Church-Rosser* Property in PTS_{atr} 79
3.2.4 Consequences of the *Church-Rosser* property 81
3.3 Equivalence of PTS_{atr} and PTSs 83
3.3.1 Confluence of the annotation process 83
3.3.2 Consequences of the Erased Confluence 86
3.3.3 Consequences of the equivalence 88

4 Formalization in *Coq* 91

4.1 Formal proof: paper or computer ? 92
4.1.1 What is a formal proof ? 92
4.1.2 Automatic resolution and induction schemes 93
4.2 Encoding PTSs in a proof assistant 95
4.2.1 Questions about encodings of binders 95
4.2.2 Higher order encodings 97
4.2.3 Our final choice: de Bruijn indices 99

IV Conclusion and Further Research 103

5 Extensions of PTS 105

5.1 Sorts, order and subtyping 107
5.2 Toward a typed conversion for CC_ω 110
5.2.1 The straightforward approach 110
5.2.2 Other attempts and possible leads 112
5.3 Other leads for future investigations 114

Part I

Introduction

Introduction

In the early days of computer programming, people were first interested in programming *effectively*, due to the limited resources that were available. Since then, the power of computers has grown in enormous proportions (and should still continue until at least 2015 according to Moore's Law), such that people are getting more and more interested in programming *safely*: during the development of a software, avoiding bugs and tracking errors can be a difficult task. Usually, programs are described by their *signature* (also known as *type*). For example, the `plus` function, that adds two natural numbers, can be given the type $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Checking that a particular program matches the required signature can prevent a lot of errors. A simple calculus to write functional programs has been introduced by Church in the 1930's, known as the *Simply Typed* λ*-Calculus*.

Lately, computer scientists have also focused their attention on the field of formal reasoning. The idea is that checking a type is not enough, one wants a formal proof that a program will compute the correct algorithm. It may seem weird to talk about computer and logical reasoning when the former is usually the problem of engineers and the latter the problem of mathematicians and philosophers. But the fact is that a logical proposition can easily be seen as a sequence of symbols, and handling such symbolic object is easily done in most programming languages. A simple example of logical statement is the identity predicate: $\forall A.A \to A$ (it should be read as "for all proposition A, A implies A"). This proposition can be intentionally interpreted as a function that, given a proof of A, computes a proof of A. There is a certain similarity between the type of `plus` and the identity predicate:

- the type of `plus` can be read as the proposition "$\mathbb{N} \times \mathbb{N}$ implies $\mathbb{N}$".
- the identity predicate can be read as the type of a program that returns an element of the same type as its input. In the λ-calculus, there is

only one function that fits this description, the *identity* function $\lambda x.x$.

In fact this connection happens at several levels, and it is known as the *Curry-Howard correspondence* [How80]. We already explained one of these levels, with the link between signatures and propositions, but we can go even further by saying that a program can be read as the proof of a proposition (and again, the other way around). This is the link between programming and proving: now we can write proofs as we would write the source code of a program, but we can also write programs by proving that their signature is a valid proposition.

This is the kind of ideas that opened the way to the development of several computer-assisted reasoning software. There are now several examples of successful proof checkers or proof assistants, like *Automath*, *LCF* and *Lego*, which are no longer used, or like more recent ones such as *ACL2*, *Agda*, *Coq*, *HOL*, *HOL-light*, *Isabelle*, *Matita*, *Mizar*, *PVS*, or *Twelf*, just to quote the most popular ones. Now it is possible to write a proof as if you were writing a program, and have their validity checked automatically by one of this software.

Most of these software implement a particular type theory [1] which are proved correct: it is trivial to build a proof checker which is able to check any statement, it just has to always return a positive answer, but such checker would not be very useful since it would be *incoherent*. This is why the theories behind these software are heavily investigated, along with any extensions of those theories. The user also needs guarantees that these software are correctly implementing the logical theory behind them, so that it is impossible to derive false propositions, like being able to derive a statement *and* its negation.

The starting point of this dissertation was an attempt to extend the expressiveness the particular theory behind *Coq*. The *Coq* proof assistant is such a tool, whose underlying theory is the *Calculus of Inductive Constructions* (or CIC for short). It is a quite rich language and its limits constantly evolve through the evolutions of *Coq*. Several stages of its evolution are well-understood (see [Wer94, Cor97, Coq]).

Our initial goal was to improve *Coq*'s unification algorithm: it is well-known that higher-order unification is undecidable [Hue75, Gol81]. Still,

[1] Which is usually an extension of the Simply Typed λ-Calculus.

there are some subcases that remain decidable and useful in practical situations. The particular subcase we were interested in was the case of *pattern unification* [Mil91, Pfe91]: by a clever use of variables, it is possible to restrict unification problems to almost fit in the first-order fragment, and thus be decidable.

Coq already tries to apply pattern unification when possible, but this unification algorithm requires the use of η-conversion, to be able to have canonical $\beta\eta$-long terms, which is currently not implemented in CIC[2]. Our first task was to investigate how we could reintroduce η-conversion inside *Coq* in a safe way.

This investigation on η-conversion (more specifically on η-expansion) made us realize that the traditional presentation of CIC, which relies on an *untyped* notion of conversion, was not an adequate framework to deal properly with η-conversion. We needed a *typed* conversion. However, the problem is that there is no real meta-theory of type systems with typed conversion if they are not normalizing systems, since no one was ever able to prove two of the most important facts about typing systems, namely *Subject Reduction* and *Confluence* for a "typed" non-normalizing system. Even if, in practice, all the logical theories used for proof assistants are normalizing (one does not want its favorite checker to loop for ever), such requirement seems really too strong to be a necessary condition for these two theorems.

Furthermore, we want to make a conservative extension of the CIC in the sense that, if we add something new to the conversion, every statement that is currently provable should still be valid in the new system. Since CIC does not rely on a typed conversion, one solution could be to build a "typed" version of CIC and prove it equivalent to our current version of CIC. By building such a translation, we would have a better framework to start thinking about η-conversion. Anything that we would be able to prove in this typed setting could be imported back into the traditional presentation of CIC. But the only way we currently know to prove such an equivalence also requires *Subject Reduction* and *Confluence* in the typed framework.

Such investigation could profit to the universal knowledge as well as other proof assistants which also could extend their theory with η-conversion. In order to avoid having to build such a meta-theory for all the implemented

[2]Even if η-conversion is part of the description of CIC in [Wer94], it has since been removed due to issue with universes subtyping and subject-reduction.

type systems one by one, we chose to work on *Pure Type Systems*, which are a framework that allows to build meta-theory for a whole *family* of type systems, and not all of them are normalizing. So, even before thinking about η-conversion, we can ask ourselves this particular question:

Are Pure Type Systems with untyped conversion equivalent to Pure Type Systems with typed conversion ?

By answering this question, we would give ourselves a solid basis to extend this result to particular implementations used in proof assistants, or to extend it by adding η-conversion. This would also allow to consider that all the semantic results we know about Pure Type Systems with typed conversion are in fact just syntactical facts, provable by means of primitive recursion instead of being consequences of the construction of a model.

This problem, first asked by Geuvers [Geu93, GW94], has been now opened for more than a decade. A partial solution has been proved by Adams [Ada06] for a subclass of Pure Type Systems known as *functional*. The main result of this dissertation is to finally give a *positive* answer to the global question: both presentations of *any* Pure Type Systems are equivalent.

There is another difficult problem concerning Pure Type Systems known as the *Expansion Postponement* problem[3]. Since we also faced this issue during our investigation on the meta-theory of Pure Type Systems, we thought our efforts towards solving this second problem were worth being written down. However, our new approach to this problem still fails to give a final answer.

The outline of this dissertation is the following:

- Chapter 1 is very simple introduction to type systems, aimed at the readers that do not have any knowledge about logic in computer science. Its purpose is to make the motivations behind the investigation of type systems a little bit clearer for those that really know nothing about it.

- Chapter 2 is about *syntactical* Pure Type Systems, systems with untyped conversion. It is focused on the investigation of the behavior of conversion, and how could we manage to improve the conversion itself. By studying a particular presentation of Pure Type Systems in sequent

[3] A complete definition of this problem is given in Chapter 2.

calculus, we give a new approach to the Expansion Postponement problem.

- Chapter 3 is about *judgmental* Pure Type Systems, systems with typed conversion. The equivalence between syntactical and judgmental Pure Type Systems is achieved by the definition of an intermediate system that is used as a bridge between both worlds.

- As an example of the power of proof assistants, this work has been completely done and checked with *Coq*. This process also ensures that no detail of the proofs have been forgotten or admitted as is: everything have been investigated and proved correct. Chapter 4 is a survey of the different techniques used to achieve such a result.

- Finally, Chapter 5 is an investigation of the possible extensions of this equivalence to more complex systems. It mainly describes our attempts at the equivalence for the *Calculus of Constructions with Universes*.

Chapter 1

Type systems in a nutshell

Contents

1.1 The general setting 10

1.2 The purpose of programming languages 11

1.3 A practical example 13

1.4 Computation on types 18

1.5 Where to go next ? 20

This chapter is a gentle introduction to type systems for those who may want to read this dissertation without a minimal background in Computer Science (hi mom, hi dad !). Anybody who already knows what λ-calculus and type theory are should not read this part and directly go to Chapter 2.

It is not meant to be rigorous nor exhaustive, and we do not aim to teach to the reader all the notions he would need to understand this work, but we rather want to explain some of the motivations behind our work to people that do not know what λ-calculus and other logical gimmicks mean, using the well-known "hand-waving explanation" method. By reading this chapter, you should at least be able to parse and read all further mathematical notations, and (we hope) to understand a bit of what we were trying to do those past three years.

1.1 The general setting

It may be unclear from the following chapters since we will go deep into obscure details, but this dissertation started from two questions about communication:

1. How do I explain to my computer what I want it to do ?

2. How do I get sure that my computer actually did what I asked ?

Computers as we know them are quite recent, with the first room-sized computers that were built during the 1940's. However, even before that, they were mechanical devices that could be informally called "computer": a Barrel organ is a musical instrument that can read and play music from a written sheet, or a Jacquard loom can create complex textiles from some recipe. Both examples are in fact using the same *input*: a collection of punched cards. As you may remember, some of the first computers also used punched cards as computing instructions or data-input. In these three cases, the goal is the same: we want to give a sequence of instructions to a device so it achieves a particular purpose, whether it be playing music or computing some complex mathematical function.

We should all agree on the following points:

- Punched cards are not a reasonable "human-readable media".

- One can not store so much information on a single punched card.

These are some reasons why we study programming languages. The ultimate language may change among people, but everyone will agree on the fact that a good programming language should be easy to write, easy to read and expressive enough to let the programmer spend most of this time "programming", not "reinventing the wheel" every time.

You may have noticed that we did not ask the language to be "computer-readable" and in fact, we *do not* want that. However, we will need another language that will fulfill this purpose, but this one does not have to be intelligible to human beings, since it is dedicated to the Computer Processing Unit (CPU). Back in the early years of computer science, both languages were almost the same (you may remember punched cards and assembly language for example), but now we can make a fortunate distinction between both. All we need is a way to translate our favorite programming language into

the favorite programming language of our CPU, which is a process called *compilation*. We will not talk about compilation anymore in this dissertation, but this is a good example to emphasize one more time our two previous questions: how can we be sure that this translation process is correct ?

1.2 The purpose of programming languages

Hopefully, we now all agree on the fact that we want to give orders to a computer in a human-friendly manner. Nowadays, computers are quite sophisticated, but they are merely bigger calculators than the ones we all used at school to compute simple equations like "what is the result of 126 divided by 3?". So programming languages should be about describing the data-structures we want to work with, and the things we want to compute with them (this is question 1 of the previous section), and also about how to compute "in a safe way" (this is question 2): we want to give orders to a computer, and have some fairly good insurances that the computed results are what we actually expected.

What do we have at hand to do such computations ? We will keep it simple by stating: a CPU which will blindly execute the orders we give him and some space for data storage, which is commonly called memory. This memory can be seen as a big array of zeros and ones (called *bits*), most of the time stored in packs of eight (eight bits are called an *octet*). In our language, we will have to handle data such as `nat`ural numbers, `string`s, `list`s and so on, but in the end, they will all be stored as a chunk of memory. So, we need a way to distinguish one chunk of memory from another: let us say we have the `plus` function that take two natural numbers and add them together. What if we call this function on a string and a list of natural numbers ? What should happen ? What will happen ?

If it happens, we can almost be sure that your program will crash. But it should never have been allowed in the first place! Just by taking a look at the types involved in such a program, we can see it fails to work. This is where question 1 and 2 are facing one another: we manage to write down a program for our computer, but it was not safe to execute.

How did we guess that this program was not safe to run ? This is an easy question: `plus` is waiting for two natural numbers, and we gave it something

else. But we are clever human beings who saw this issue, while the computer only saw two chunk of memory with octets in them. It is our work, through the programming language, to give more information to the computer, so it can avoid such traps. We need to make the computer understand that a chunk of memory representing a string should not be used when a natural number is expected. This is commonly achieved by adding *labels* to this structure, saying for example "this chunk is a natural number and this other one is a list of strings". With this information, the computer can be aware of the kind of data it is computing over, and will reject a program like `plus (3,"foo")` just because `"foo"` is not a natural number.

These labels are called *types*, and the action of seeing if the type of an input matches the expected type of a function is called *type checking*. The structure of types inside a programming language is called a *type system*. They are here to enforce some guard properties in a program, to have some guarantees that the computation will be done in a safe manner. The expressiveness of type systems is directly linked to the kind of guarantees we will have: if they are too simple, we will only be able to have simple information as in the toy example we just saw. With more expressive languages, we can have much more powerful assertions, like pre-conditions to fulfill before being able to use a function, or additional information on the values returned by a program.

A common example to emphasize the power of a type system is Euclid's division algorithm: extensionally, it can be stated as "given a and b two natural numbers, if b is not equal to 0, we want to find q and r such that $a = b * q + r$ and $r < b$". In real programming languages, the type of this program may vary depending of the power of the type system:

- in C-like: `void div(int a, int b, int *q, int *r)`
- in ML-like: `div: int -> int -> (int * int)`
- in Coq: `div : forall (a b:nat), 0 < b -> {q : nat & {r : nat & a = b * q + r /\ r < b }}.`

In the first two examples, the programming language is not informative enough to carry around information "about" the data, like b is not 0 or $a = b * q + r$. The only information we have is that `div` is expecting two natural numbers (or `int`egers) and will return two natural numbers, without any information about the computation itself.

In the last one, there is a pre-condition and two post-conditions: to be able to call this function, we need to give a witness that b is not 0, and this function will give us two natural numbers that verify the two relations that we are interested in. It is pretty nice to have such power in the type system, but there is a drawback: in the code of the third program, we need to build the proof that the resulting q and r enjoy the nice relations with the input. More expressiveness will require more work from the programmer. In this particular example, we already know some automatic ways to prove these arithmetical results, but we have to keep in mind the following statement: the more we want, the more we need to provide first.

1.3 A practical example

To be able to study a programming language, we first need to explain its syntax, and then we can try to prove some nice properties over it. In fact, we need to define two things: the syntax of the language, and the process of computation. We need to *formalize* our language along with the process of computation. As an example of the properties a programming language can have, we are going to consider a simple (but still powerful) example by studying the *λ-calculus*.

This language was introduced by Alonzo Church [Chu51, Bar84] in the 1930's as a tool to study the foundations of mathematics. Since then, many extensions of this language have been used as a basis for real programming language and for logic programming [HAS, SML, Gal, PS]. Here is its syntax:

$$M, N \quad ::= \quad x \mid M\ N \mid \lambda x.M$$

A term of the λ-calculus can be built from three different constructors. The first one is about *variables* (which we will always write as lower-case letters x, y, ...). They are names for more complex terms. The second one, $M\ N$, is called an *application*: given a function f and an argument a, $f\ a$ stands for "let us apply the function f to a". Back in high-school, we would write it $f(a)$, this is just another notation for it. Please remember that this is *not* the result of the computation of this function when applied to a particular argument, it is just the syntactic juxtaposition of two terms: for example, if we consider the identity function `id` that just return whatever data we give it, `id` a is syntactically different from a, but it will compute to a. If a function has several arguments (like the `plus` function for example),

we simply use several applications :(`plus` x) y. By convention, application is left-associative, so we can even get rid of the parentheses and just write `plus` x y.

Finally, the most awkward of the symbols above, the λ-abstraction is used to define functions. Until now, we gave name to functions but names can be complicated to handle during a strict formalization, so we introduce *anonymous* functions with this abstraction. Here are some simple examples to illustrate this λ construction:

- the identity function is denoted by the term $\lambda x.x$, which is equivalent to the declarative statement "for any term x, we return x".
- the `plus` function[1] is denoted by the term $\lambda x \lambda y.x + y$: "given a x and a y, we return $x + y$".
- the `app` function, that takes a function, an argument, and apply this function to this argument is denoted by the term $\lambda f \lambda x.f\ x$.

In a λ-abstraction $\lambda x.M$, the variable x is *bound* by the λ inside the body M. It means that if we apply this function to an argument N, all occurrences of x inside M will be placeholders for N.

Now that we have a simple syntax for terms (without types at the moment), we need to explain how to compute with them: how do we go from `id` x to x ? This process of rewriting a term into another one is called β-reduction, and is informally defined as follows: if an abstraction $\lambda x.M$ is applied to term N, the application $(\lambda x.M)\ N$ can be *β-reduced* to $M[N/x]$, which stands for "replace all the occurrences of x in M by N". This process will be noted $\rightarrow_\beta$ for a one step reduction, and $\twoheadrightarrow_\beta$ for several consecutive steps:

- $\texttt{id}\ N \rightarrow_\beta N$
- $\texttt{app id}\ a \rightarrow_\beta (\lambda x.\texttt{id}\ x)\ a \rightarrow_\beta \texttt{id}\ a$
- $\texttt{plus 3 4} \twoheadrightarrow_\beta \texttt{7}$

[1] Here we cheated: `+` is not part of our syntax, but it was just to illustrate with a known example.

This language is rather simple, but it is quite interesting to study its computational behavior, even without type information: we can encode a lot of interesting data-structures (natural numbers, pairs, lists, ...), but this is not what we are interested in at this point.

What can we say about this language ? What properties does it give us on the structure of our program, or on its computation behavior? In fact, nothing much at the moment: we need types! Since we only have one function constructor (using the λ-abstraction), we only need one type constructor, written $\rightarrow$, which is pretty much like the mathematical implication for function:

- `id` has type $A \rightarrow A$: it takes a data of type A and returns it directly, so the type is not changed.
- `plus` has type `nat` $\rightarrow$ `nat` $\rightarrow$ `nat`: it takes two natural numbers and returns a natural number.
- `app` has type $(A \rightarrow B) \rightarrow A \rightarrow B$: it takes a function of type $A \rightarrow B$, an argument of type A, and returns the result of their application, which is of type B.

We will also need some *base types* (like `nat` or `string`) for our data, but we do not really care about it for our study. Last thing, we need to store all the type information we will collect in a *context*, as a remainder of the types that we already know: as soon as we know that `plus` has type `nat` $\rightarrow$ `nat` $\rightarrow$ `nat`, we can put this information in our context so we do not need to rebuild this information the next time we will need it.

With all this, we can build a type system for the λ-calculus, called the *Simply Typed λ-calculus*. This extension of the λ-calculus was also introduced by Church [Chu40] a few years after first presenting the λ-calculus to avoid some paradoxes that were discovered in the untyped calculus. The system is defined according to the following rules:

- a variable x has type A if this information is in our context.
- if, knowing that x has type A, we can prove that M has type B, then $\lambda x.M$ has type $A \rightarrow B$.
- if M is a function of type $A \rightarrow B$ and N has type A, the resulting application $M\ N$ has type B.

Since we are trying to be a little more formal, we now give the *same* set of rules, but written in a much more concise way, it will be easier to talk about a specific rule. The typing rules for the *Simply Typed λ-calculus* are detailed in Fig. 1.1. This kind of presentation is called a *type system*.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \textsc{var} \qquad \frac{\Gamma(x : A) \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}\ \textsc{lam} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \textsc{app}$$

Figure 1.1: The Simply Typed λ-Calculus

This presentation is really simple to read: a typing judgment (also called a sequent) $\Gamma \vdash M : A$ means that in the context Γ, the term M has type A, and the lines can be read as a simple implication:

$$\frac{\textit{If all the conditions on top are true}}{\textit{the conclusion at the bottom is also true}}\ \textsc{name of the rule}$$

You may have noticed that, in the LAM rule, we wrote $\lambda x : A.M$ instead of $\lambda x.M$. This additional annotation is quite handy to reason about typing judgments, but it is not mandatory: both presentations, with or without the annotation on λ-abstractions, are commonly used. For the application rule, we take care to check that the type of the argument N matches the type expected by the function M: they are both the same A.

A well-typed term is a term for which we can compute a type using only the previous rules: `plus 3 4` is a well-typed term because `3` and `4` are natural numbers, but `plus 3 "foo"` is not because `"foo"` is a string, so we will not be able to correctly apply the APP rule.

With this nice syntax and its simple type system, we can start proving properties about our programming language. We are going to focus on two properties which are, according to us, the most important ones: *Termination* and *Preservation*.

A term is said to be terminating if its computation terminates, that is if we can only apply a finite number of β-reduction steps: our running example `plus 3 4` is terminating, because `7` can not be reduced anymore. This property is used to ensure there is no *infinite loop* inside a program. As you may guess, all the terms are not terminating (otherwise we would not have to state such a property). However, all the *well-typed terms* are. Take a look

at the term $\lambda x.x\ x$, that we will call δ. Without the typing information, we can do some reductions:

$$\begin{array}{rcl} \delta\ \delta & = & (\lambda x.x\ x)\ \delta \\ & \rightarrow_\beta & (x\ x)[\delta/x] = \delta\ \delta \\ & \twoheadrightarrow_\beta & \delta\ \delta \\ & \twoheadrightarrow_\beta & \ldots \end{array}$$

Having such an infinite reduction sequence can be a bit annoying inside a program, especially when we want to compute a result: it will loop forever! But what if we try to attach *types* to this term ? Let us try to guess the type of δ, with the previous rules:

1. δ is a λ-abstraction, so its type is of the shape $T_1 \rightarrow T_2$, and in a context where x has type T_1, $x\ x$ has to have type T_2.

2. the first occurrence of x forces the shape of T_1: $T_1 = A \rightarrow B$.

3. the second one forces the type of x to be equal to the domain of T_1, so we get another equation : $T_1 = A$.

All this leads to a single equation $A = A \rightarrow B$ which is unsolvable: δ is not typable in the Simply Typed λ-Calculus. This example shows how typing can reject non-terminating programs, without having to run them.

The second property we want to emphasize is *Preservation* (also known as *Subject Reduction*). It states that computation does not change the type of a term: if $M \rightarrow_\beta N$ and M is of type A, then N is also of type A. In a more formal presentation:

$$\frac{\Gamma \vdash M : A \qquad M \rightarrow_\beta N}{\Gamma \vdash N : A}$$

This property ensures that computation does not mess with the content of the data: for example, just from the typing information we have so far, we know that `plus 3 4` is a natural number, because we already know the type of `plus`, and because `3` and `4` are natural numbers, but preservation guarantees that the resulting computation, `7`, is also a natural number, even if we do not perform this computation. In this case, it was easy to check that `plus 3 4` will compute to a natural number, but it is not always so simple to verify.

1.4 Computation on types

A nice property about the Simply Typed λ-Calculus is that it has literally simple types: the world of types and the world of terms are completely distinct one from the other. With this kind of system, it is impossible to have complicated information in the types: if you recall the example about Euclid's algorithm, the third one was the most informative one, but there were actually *terms* (namely a, b, q and r) inside the *type* of the function.

In order to achieve such expressiveness, we need to add a dependency between types and terms. There are several ways to do this, at several levels (types depending on types, types depending on terms, terms depending on types, ...) and we do not want get into all of them at the moment. These systems are significantly more difficult to understand than the simple λ-calculus, so we are not going to try to describe a particular one, but we are just going to think about one question : why would we want to compute inside a type?

The usual running example at this point is to consider a particular kind of lists. The basic type for list of terms has only one parameter: the type of the data it contains:

- the list $l_1 = [\texttt{1};\texttt{2};\texttt{3};\texttt{4};\texttt{5}]$ has type `list nat`.
- the list $l_2 = [\texttt{'a'};\texttt{'b'};\texttt{'z'}]$ has type `list char`.
- a list containing data of type `A` has type `list A`.

We are going to extend this definition by adding a second information in the type: we want to know the *size* of the list just by typing, and will call this new type `nlist`:

- l_1 would have type `nlist 5 nat`.
- l_2 would have type `nlist 3 char`.
- a list of length `n` containing data of type `A` has type `nlist n A`.

This new type allows us to be more informative while writing a function. For example, if we want to extract the first element of a list, we need to check that this list is not empty: with `nlist`, this can be done with typing !

```
head :  forall (A:Type) (n:nat), nlist (n+1) A -> A.
```

The `head` function expects a list of length *at least 1*, so its execution will never involve the empty list. This is a pretty interesting feature, but what if we go with more complicated functions ? Let us consider the concatenation of two lists:

```
concat:
    forall (A:Type) (n m:nat), nlist n A -> nlist m A -> nlist
(n+m) A.
```

The `concat` function takes two lists of arbitrary size, `n` and `m`, and return a list which size is `n + m` by gluing the second list at the end of the first one. A practical example: what is the result and the type of `concat [1;2] [3;4;5]` ?

```
concat [1;2] [3;4;5] = [1;2;3;4;5] :  nlist (2+3) nat.
```

Why is it `2+3` and not `5` ? Simply because, as we previously said, `2+3` is not *syntactically* equal to `5`, but it computes to `5`. If we want to embed terms into types, we also need a way to embed the computation at the level of types. The usual way to do this is to add a *conversion* rule to our favorite type systems, which looks like:

$$\frac{\Gamma \vdash M : A \qquad A \textit{ computes to } B}{\Gamma \vdash M : B}$$

The critical notion here is how to define the “computes to”. Several different presentations have been proposed until now, all designed for a particular purpose:

- If one is only concerned with *program evaluation*, one only needs to have an *untyped* notion of reduction, and rely on *preservation* to type the result of its computation.

- If one is concerned with *consistency*, or if its computation needs type (it is the case with η-expansion for example), one may need to have a *typed* notion of reduction, but will have trouble to prove *preservation*.

The following chapters are a technical investigation about this conversion rule, in order to finally prove that all the definitions that we study in this dissertation are actually just different ways to state the *very same* things: all these presentations are equivalent.

1.5 Where to go next ?

In this first chapter, we wanted to explain as simply as possible the underlying motivations of our work: why are we interested in type theory, and what are the possible applications of this field. From now on, we will forget a bit about computer programming, and focus on the study of the meta-theory of a particular family of type systems called Pure Type Systems, which are used as a basis for the underlying theory of proof assistants and proof search engines.

Our work will mainly focus on the conversion rule of those systems which is the main reason why there are several different presentations of those systems, depending on the kind of guarantees one wants about the computation.

This investigation aims to improve our understanding of the theories behind proof assistants, in order to improve those software. As a practical application of this concept of "proof assisted by computer", this dissertation has been completely formalized within the *Coq* proof assistant [Coq], as a supporting tool which helped to build some complex parts of the proofs, and also to be sure that we did not forget anything in the formalization, and thus ensure that everything was correctly proved. The whole development can be found here [Sila, Silb] and has been tested with both the *trunk* version of July 2010 (revision 13303) and the 8.3 version of *Coq*.

Part II

Untyped equality

Chapter 2

Pure Type Systems

Contents

2.1 Pure Type System in Natural Deduction 25

2.1.1 Terms and Untyped Reductions 25

2.1.2 Presentation of Pure Type Systems 28

2.2 Pure Type Systems in Sequent Calculus 36

2.2.1 Terms and Reduction 38

2.2.2 Confluence of β-reduction in $\overline{\lambda}$ 39

2.2.3 Typing Rules 42

2.2.4 Properties of the system 42

2.3 Delayed Pure Type System in Sequent Calculus 45

2.3.1 Typing Rules 46

2.3.2 Properties of the system 46

2.4 Expansion Postponement in Delayed System . . 50

2.4.1 Postponement in Sequent Calculus 51

2.4.2 Postponement in Natural Deduction 52

2.5 Sequent Calculus and Type Inference 54

2.6 A brief look back at syntactical Pure Type Systems 57

The first chapter was a first glimpse at type systems: the study of these systems can give a lot of information on programming languages. However,

it can be really fastidious to study *every* type system that exists, one at a time. From now on, we are going to focus only on functional languages, and especially on a particular framework known as *Pure Type Systems* (or PTSs from now on). This framework allows to describe and study a large family of type systems by considering only a single one which relies on some abstract parameters. Doing so, we can study all those systems at once by proving properties of the abstract system, properties that will automatically be inherited by all the instances of this system. In the end, we are able to select one particular system by simply providing the correct parameters to the system.

Pure Type Systems were first introduced independently by Berardi and Terlouw, mainly inspired by Barendregt's λ-cube [Ber88, Bar92]. The purpose of this cube was to classify the different ways terms and types may depend on each other in some well-known type systems, from simple system like the *Simply Typed λ-Calculus* to more complicated ones like the *Calculus of Constructions*. As we previously said, when there is a dependency of types over terms, we need a way to compute inside those types. This particular computation will grab most of our attention in the upcoming chapters.

The reason is that, in practice, there is not a unique presentation of those Pure Type Systems, they exist with different flavors for the shape of the rules, or even for the notion of conversion. The main result of this work is to prove that most of the interesting presentations are actually equivalent: they all describe the very same theory. With this result, one can chose the best of every presentation without any restriction.

In this chapter, we give an overview of the declarative presentation of PTSs (also known as syntactical PTSs), whose conversion rule is based on an external notion of equality: conversion does not depend on typing. They are already quite well understood, so we are only going to recall here the major properties and the most difficult proofs, but we also want to highlight the mechanics of their meta-theory. By understanding the order in which things can be proved, and how they are used, it will be easier to understand the more complicated problems of their typed counterpart with judgmental equality.

The first section is dedicated to the usual presentation, based on natural deduction. It will be used as a basis in the following chapter to deal with typed equality. In the second section however, we present a variant of

those Pure Type Systems (mainly inspired by Lengrand's thesis [Len06]) by switching from natural deduction to sequent calculus, in order to address a particular open question about PTSs called Expansion Postponement.

2.1 Pure Type System in Natural Deduction

As we said, this framework relies on an *external* notion of equality, which does not mix with typing. From now on, we will consider this equality to be the β-conversion. The order in which we will state the following properties is relevant, it will help to understand the issues that will arise in the next chapter.

In this section, we are going to only focus on the traditional presentation of PTSs, in natural deduction. We first present the underlying terms and reduction used to describe these PTSs, then we explain the formal definition of PTSs along with some main results. The last part is dedicated to a particular result often unknown, which is a lead to some of the most difficult results about PTSs, like *Strengthening*.

2.1.1 Terms and Untyped Reductions

The terms used in the following type systems are the usual λ-calculus terms *à la* Church — variable, annotated abstraction and application — extended with two more constructions which are the entry points of types inside terms : Π-types and sorts.

Structure of terms and contexts
$s \;:\; Sorts$
$x \;:\; Vars$
$A, B, M, N ::= \; s \mid x \mid MN \mid \lambda x^A.M \mid \Pi x^A.B$

$\Gamma \; ::= \; \emptyset \mid \Gamma(x : A)$

The syntactical equality between two terms M and N is written $M \equiv N$.

As you can see, there is no syntactical distinction between terms and types, they are both part of the same grammar. Since we want to cover the whole generality of dependent type systems (especially those parts of the λ-cube), we need to be able to handle any kind of dependency, such as types depending

on types (*System F*), types depending on terms (*Calculus of Constructions*) or even no dependency at all (*Simply Typed λ-Calculus*). Using the same syntax for terms and types avoids redefining several times the same binders for all those kinds of dependencies. The separation is done by the typing rules afterwards.

The Π construct will be used to type functions, and $\Pi x^A.B$ is usually noted $A \to B$ when B does not depend on its argument x. If there is a dependency, we keep track of the binding variable x with the full notation.

The set *Sorts* is the first parameter that defines an instance of a PTS. Sorts are used to assert that a term can correctly be used in a typing position. We will see how it works in more details after the introduction of the typing rules. The set of variables *Vars* is assumed to be infinite, and is common to all PTSs. In the following, we consider s, s_i and t to be in $Sorts$, and x, y and z to be in $Vars$. A context is a list of terms labeled by distinct variables, e.g. $\Gamma \equiv (x_1 : A_1) \dots (x_n : A_n)$, where all the x_i are distinct. Since we want to handle dependent types, the order inside the context matters: a x_i can only appear in A_j where $j > i$. For convenience, we will often use the notation $\Gamma(x) = A$ as a shortcut for $(x : A) \in \Gamma$ (a context can be seen as a finite map from $Vars$ to terms, where the orders of the keys matters) and $\emptyset$ denotes the empty context. The *domain* $Dom(\Gamma)$ of a context Γ is defined as the set of x_i such that $\Gamma(x_i)$ exists. The concatenation of two contexts whose domains are disjoint is written $\Gamma_1\Gamma_2$.

The term $\lambda x^A.M$ (resp. $\Pi x^A.B$) *binds* the variable x in M (resp. B) but not in A and the set of *free variables* (fv) is defined as usual according to those binding rules:

$$
\begin{array}{lcl}
\text{-}\ fv(s) & \triangleq & \{\} \text{ if } s \text{ is a sort.} \\
\text{-}\ fv(x) & \triangleq & \{x\} \text{ if } x \text{ is a variable.} \\
\text{-}\ fv(MN) & \triangleq & fv(M) \cup fv(N). \\
\text{-}\ fv(\lambda x^A.M) & \triangleq & fv(A) \cup (fv(M) \backslash \{x\}). \\
\text{-}\ fv(\Pi x^A.B) & \triangleq & fv(A) \cup (fv(B) \backslash \{x\}).
\end{array}
$$

We use an external notion of substitution: $_[_/_]$ is the usual function of substitution, and $M[N/x]$ stands for the term M where all the occurrences of the free variable x have been replaced by N, without any variable capture (in order to avoid any collision between x and a bound variable in M, we can always α-rename the bound variables in M to fresh ones). We can extend

the substitution to contexts (in this case, we consider that $x \notin Dom(\Gamma)$). $\Gamma[N/x]$ is recursively defined as :

1. $\emptyset[N/x] \triangleq \emptyset$
2. $(\Gamma(y : A))[N/x] \triangleq \Gamma[N/x](y : A[N/x])$

Now that we have defined the syntax for terms, it is time to take a closer look at the conversion process we are going to use. The notion of β-reduction ($\rightarrow_\beta$) is defined as the congruence closure of the relation $(\lambda x^A.M)N \rightarrow_\beta M[N/x]$ over the grammar of terms. The reflexive-transitive closure of $\rightarrow_\beta$ is written as $\twoheadrightarrow_\beta$, and its reflexive-symmetric-transitive closure as $=_\beta$ (which can be called *conversion* or *untyped equality* in the following). The notion of syntactic equality (up to α-conversion) is denoted as $\equiv$.

A main property of β-reduction is *Confluence*:

Theorem 2.1.1 (*Confluence* of β-reduction)**.** *For all terms M, N and P, if $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta P$, then there is Q such that $N \twoheadrightarrow_\beta Q$ and $P \twoheadrightarrow_\beta Q$.*

Proof. There are several well-known proofs of *Confluence* in the literature, but we would like to emphasize the one using parallel reduction (see [Bar84] for the usual proof of *Tait Martin-Löf*, or [Tak95] for another way to prove it).

Using β-reduction, one can reduce *exactly one* redex per reduction step. The idea behind the parallel reduction ($\rightarrow_p$) is to allow the reduction of *all* redexes that appears in separate subterms (thus the name of "parallel")[1] along with the *reflexivity* property that β-reduction is lacking to enjoy the diamond property.

For example, here are the rules for the application and head-reduction cases:

$$\frac{M \rightarrow_p M' \qquad N \rightarrow_p N'}{MN \rightarrow_p M'N'} \qquad \frac{M \rightarrow_p M' \qquad N \rightarrow_p N'}{(\lambda x^A.M)N \rightarrow_p M'[N'/x]}$$

In the first rule, we are allowed to reduce all the redexes that appear in both part of the application. Because they are two different subterms, no reduction in one of them can make a new redex appears in the other one. However, in the second case, the head reduction is going to mix both subterms, so we are only allowed to reduce *one* redex in head position at a time.

[1] This is a particular case of *finite development* [Bar84].

Such a reduction enjoys the *Diamond Property*, and also has the same transitive closure as the usual β-reduction. Those two properties are enough to show that β-reduction is *Confluent*. $\square$

At this point, it is important to notice the order in which we can prove things: *Confluence* of the β-reduction can be established before even defining the typing system, it is only a property of the reduction. Using this, we can prove some useful properties of Π-types and sorts:

Lemma 2.1.2 (Consequences of Confluence)**.**

- Π-injectivity*: If* $\Pi x^A.B =_\beta \Pi x^C.D$ *then* $A =_\beta C$ *and* $B =_\beta D$
- *If* $s =_\beta t$ *then* $s \equiv t$*.*
- *The statement* $\Pi x^A.B =_\beta s$ *does not hold for any* A, B *and* s*.*

2.1.2 Presentation of Pure Type Systems

As we said, a PTS is a generic framework that allows us to study a family of type systems all at once. Popular type systems like *Simply Typed λ-Calculus*, *System F* , the systems U and U^- or the *Calculus of Constructions* are part of this family. They have been brought mainstream by Barendregt in the early 1990's, and were also known as *Generalized Type Systems*. There is plenty of literature on the subject [Bar92, vBJMP93] so we will only recall the main ideas of those systems.

The abstract nature of a PTS arise in the typing rules for sorts and Π-types. The set $\mathcal{A}x \subset (Sorts \times Sorts)$ is used to type sorts: $(s,t) \in \mathcal{A}x$ means that the sort s can be typed by the sort t. The set $Rel \subset (Sorts \times Sorts \times Sorts)$ is used to check the well-formedness of Π-types.
The typing rules for PTSs are given in Fig. 2.1. Intuitively, $\Gamma \vdash M : T$ can be read as "the term M has type T in the context Γ", and $\Gamma \vdash A : s$ as "A is a valid type in Γ". In the following, even if terms and types share the same syntax in PTSs, we will often call *type* a term which is typed by a sort.

As we can see, the CONV rule relies on the external notion of β-conversion, so we do not check that every step of the conversion is well-typed. However, it is easy to prove *Confluence* and *Subject Reduction*, two properties which ensure that everything goes well.

$$\frac{}{\emptyset_{wf}}\ \text{NIL} \qquad \frac{\Gamma \vdash A : s \qquad x \notin Dom(\Gamma)}{\Gamma(x:A)_{wf}}\ \text{CONS}$$

$$\frac{\Gamma_{wf} \qquad (s,t) \in \mathcal{A}x}{\Gamma \vdash s : t}\ \text{SORT} \qquad \frac{\Gamma_{wf} \qquad \Gamma(x) = A}{\Gamma \vdash x : A}\ \text{VAR}$$

$$\frac{\begin{array}{cc}\Gamma \vdash A : s & \Gamma(x:A) \vdash B : t \\ (s,t,u) \in \mathcal{R}el & \Gamma(x:A) \vdash M : B\end{array}}{\Gamma \vdash \lambda x^A.M : \Pi x^A.B}\ \text{LAM} \qquad \frac{\begin{array}{c}\Gamma \vdash A : s \\ \Gamma(x:A) \vdash B : t \qquad (s,t,u) \in \mathcal{R}el\end{array}}{\Gamma \vdash \Pi x^A.B : u}\ \text{PI}$$

$$\frac{\Gamma \vdash M : \Pi x^A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}\ \text{APP} \qquad \frac{\begin{array}{c}A =_\beta B \\ \Gamma \vdash M : A \qquad \Gamma \vdash B : s\end{array}}{\Gamma \vdash M : B}\ \text{CONV}$$

Figure 2.1: Typing Rules for PTSs

Since we only require *Sorts* to be a set, one can choose any labeling system that suits ones need to name the sorts. There is an informal habit to choose symbols like $\star$ or $\square$ when there are few sorts, or indexed names like $Type_i$ or $\square_i$ when this set is infinite. Here are two simple examples (from the λ-cube) of such instantiation of PTSs:

Definition Examples of PTSs

- the *Simply Typed λ-Calculus*:

$$\begin{array}{rcl} Sorts &=& \{\star, \square\} \\ \mathcal{A}x &=& \{(\star, \square)\} \\ \mathcal{R}el &=& \{(\star,\star,\star)\} \end{array}$$

- the *Calculus of Constructions*

$$\begin{array}{rcl} Sorts &\triangleq& \{Prop, Type\} \\ \mathcal{A}x &\triangleq& \{(Prop, Type)\} \\ \mathcal{R}el &\triangleq& \{(s, Prop, Prop), (Prop, s, s) \mid s \in Sorts\} \cup \\ && \{(Type, Type, Type)\} \end{array}$$

Remark The traditional way describe PTSs can be found in [Bar92]. Here, I presented here a version of the typing rules inspired by [vBJMP93], which is designed to help the formalization of this meta theory in a proof assistant. The proof of equivalence of these two presentations is easily done by induction and some basic properties of PTSs we are going to introduce (2.1.6, 2.1.7).

Definition Top sorts
A sort $s \in Sorts$ is called a *top sort* if it never appears at the right-hand side of a pair in $\mathcal{A}x$, i.e. there is no t such that $(s,t) \in \mathcal{A}x$. In the previous examples, $\square$ and $Type$ are top sorts, whereas $\star$ and $Prop$ are not.

The following properties hold for all PTSs. Even if they are quite technical, they are the basic meta-theory that we need to prove the interesting theorems.

The two following lemmas are the first structural properties of PTSs: *Weakening* allows to add more assumptions in the context without changing the typing judgment, and *Substitution* witnesses the correct use of the external substitution (substitution is allowed if the types match).

Lemma 2.1.3 (Weakening)**.**

1. *If* $\Gamma_1\Gamma_2 \vdash M : B$, $\Gamma_1 \vdash A : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2 \vdash M : B$.
2. *If* $\Gamma_1\Gamma_2\ _{wf}$, $\Gamma_1 \vdash A : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2\ _{wf}$.

Lemma 2.1.4 (Substitution)**.**

1. *If* $\Gamma_1(x : A)\Gamma_2 \vdash M : B$ *and* $\Gamma_1 \vdash P : A$ *then* $\Gamma_1\Gamma_2[P/x] \vdash M[P/x] : B[P/x]$.
2. *If* $\Gamma_1(x : A)\Gamma_2\ _{wf}$ *and* $\Gamma_1 \vdash P : A$ *then* $\Gamma_1\Gamma_2[P/x]_{wf}$.

While proving facts about PTSs, we will often need to compute some typing information about the subterms of one judgment. To do this, we will frequently use the *Generation* (or *Inversion*) property and the *Validity of Contexts*:

Lemma 2.1.5 (Validity of Contexts)**.**
For all Γ, M *and* T, *if* $\Gamma \vdash M : T$, *then* Γ_{wf}.

Theorem 2.1.6 (Generation)**.**

1. *If* $\Gamma \vdash s : T$ *then there is* t *such that* $(s,t) \in \mathcal{A}x$ *and* $T =_\beta t$*.*
2. *If* $\Gamma \vdash x : A$ *then there is* B *such that* $\Gamma(x) = B$ *and* $A =_\beta B$*.*
3. *If* $\Gamma \vdash \Pi x^A.B : T$ *then there are* s_1, s_2, s_3 *such that* $\Gamma \vdash A : s_1$*,* $\Gamma(x : A) \vdash B : s_2$*,* $(s_1, s_2, s_3) \in \mathcal{R}el$ *and* $T =_\beta s_3$*.*
4. *If* $\Gamma \vdash \lambda x^A.M : T$ *then there are* s_1, s_2, s_3 *and* B *such that* $\Gamma \vdash A : s_1$*,* $\Gamma(x : A) \vdash B : s_2$*,* $\Gamma(x : A) \vdash M : B$*,* $(s_1, s_2, s_3) \in \mathcal{R}el$ *and* $T =_\beta \Pi x^A.B$*.*
5. *If* $\Gamma \vdash M\ N : T$ *then there are* A *and* B *such that* $\Gamma \vdash M : \Pi x^A.B$*,* $\Gamma \vdash N : A$ *and* $T =_\beta B[N/x]$*.*

As we previously said, terms and types can only be distinguished by their use in the typing rules: in $\Gamma \vdash M : T$, M is a “term” and T is a “type”, and M is of type T, but one can wonder what the type of T is.

Lemma 2.1.7 (Type Correctness)**.** *If* $\Gamma \vdash M : T$*, then there is* s *such that* $T \equiv s$ *or* $\Gamma \vdash T : s$*.*

We certainly would like to always be able to prove that T is well-typed, but because of the generality behind PTSs, and especially because of the definition of $\mathcal{A}x$, we can not guarantee this. There are some sorts that can not be typed, that is any sort which do not appear in left position of any pair in $\mathcal{A}x$ (the *top sorts* that we previously introduced), like $Type$ and $\Box$.

In the next chapter, we will often refer to two particular subclasses of PTSs which enjoy some interesting properties about their types: the *functional* and the *semi-full* PTSs.

Functional, Full and semi-Full PTSs

- A PTS is functional if:
    1. for all s, t, t', $(s,t) \in \mathcal{A}x$ and $(s,t') \in \mathcal{A}x$ implies $t \equiv t'$.
    2. for all s, t, u, u', $(s,t,u) \in \mathcal{R}el$ and $(s,t,u') \in \mathcal{R}el$ implies $u \equiv u'$.
- A PTS is semi-full if $(s,t,u) \in \mathcal{R}el$ implies that for all t', there is u' such that $(s,t',u') \in \mathcal{R}el$.

- A PTS is full if for any s, t, there is u such that $(s, t, u) \in \mathcal{R}el$.

  Obviously, a full PTS is also semi-full.

Let first take a loot at the *functional* class. All systems within Barendregt's λ-*cube* can be seen as functional PTSs, like the *Simply Typed λ-Calculus*, or the *Calculus of Constructions*. Being functional implies that the sorts have a unique type, and this property can be extended to the whole system:

Lemma 2.1.8 (Type Uniqueness for functional PTSs)**.**
In any functional *PTS, if* $\Gamma \vdash M : T$ *and* $\Gamma \vdash M : T'$ *then* $T =_\beta T'$.

Proof. By induction and generation, all the cases are trivial except for SORT and PI, but for them, the hypothesis of functionality allows to conclude directly. □

This property is somehow useful and quite appealing: a term only lives in a unique type. However, this restriction forbids us to add subtyping to our system, and explore systems such as the *Extended Calculus of Constructions* or the *Calculus of Inductive Constructions*.

The second interesting subclass is the *semi-full* fragment of PTSs. This class has been first defined by Pollack [Pol92, Pol94] in order to find an subclass of PTSs where type-checking was possible. Here we are interested in this class (which is a bit more general than the class of full PTSs) because, unlike the functional class, we can extend those systems with a subtyping relation on sorts.

The *Calculus of Constructions* is also part of this class, but not the *Simply Typed λ-Calculus*. It is easier to first give an intuition about the *full* subclass: in a *full* PTS, all the products are allowed, as soon as their domain and co-domain are well-typed. For *semi-full*, the idea is a bit more subtle: if a product $\Pi x^A.B$ is well-typed, then any product $\Pi x^A.D$ is well-typed as soon as D is well-typed. We will call this property *the functionality of* Π-*types*.

Let us go back to the general picture. The notion of β-reduction and conversion can easily be extended to contexts since they are ordered lists of terms

Context Reduction and Context Conversion

1. Reduction :
   - If $A \rightarrow_\beta B$ and $x \notin Dom(\Gamma)$, then $\Gamma(x : A) \rightarrow_\beta \Gamma(x : B)$.
   - If $\Gamma \rightarrow_\beta \Gamma'$ and $x \notin Dom(\Gamma)$, then $\Gamma(x : A) \rightarrow_\beta \Gamma'(x : A)$.
2. Conversion:
   - $\emptyset =_\beta \emptyset$.
   - If $\Gamma =_\beta \Gamma'$, $A =_\beta B$ and $x \notin Dom(\Gamma)$, then $\Gamma(x : A) =_\beta \Gamma'(x : B)$.

Lemma 2.1.9 (Context Conversion in Judgments)**.**
If $\Gamma \vdash M : A$, $\Gamma =_\beta \Gamma'$ *and* Γ'_{wf} *then* $\Gamma' \vdash M : A$.

We still need to check that Γ' is well-formed, even if we only do reductions, because we do not have *Subject Reduction* at hand yet.

Along with the CONV rule, we have now some kind of liberty with our types: we can change any type with a convertible one (and, of course, well-formed) in the context or in typing position. With all those tools, we can now prove the main property of PTSs, which states that computation preserves typing:

Theorem 2.1.10 (Subject Reduction)**.** *If* $\Gamma \vdash M : A$ *and* $M \rightarrow_\beta N$, *then* $\Gamma \vdash N : A$.

Proof. The proof can be found in [Bar92]. We just want to put forward that it relies on *Confluence*, more precisely on the Π-*injectivity* of β-reduction for the case where M is a redex $(\lambda x^{A'}.P)N$ which reduces to $P[N/x]$: by induction, we retrieve typing information about both subterms of M: $\Gamma \vdash N : A$ and $\Gamma \vdash \lambda x^{A'}.P : \Pi x^A.B$. Our goal is to show that $\Gamma \vdash P[N/x] : B[N/x]$. Thanks to *Generation* and *Type Correctness*, we can fetch information about A, A', B and M:

- there are $s, t, u \in Sorts$ such that $\Gamma \vdash A : s$, $\Gamma(x : A) \vdash B : t$ and $(s, t, u) \in \mathcal{R}el$.
- there are $s', t', u' \in Sorts$ and B' such that $\Gamma \vdash A' : s'$, $\Gamma(x : A') \vdash B' : t'$, $\Gamma \vdash (x : A') \vdash P : B'$ and $(s, t, u) \in \mathcal{R}el$.

- $\Pi x^A.B =_\beta \Pi x^{A'}.B'$.

We can not directly apply the *Substitution* Lemma for two reasons: N is of type A and not A', and P is of type B', not B. However, *thanks to* Π-*injectivity*, we have that $A =_\beta A'$ and $B =_\beta B'$, so we just need to use CONV and *Context Conversion* to be able to apply *Substitution*. □

Now that we have *Subject Reduction*, we can prove that any use of the CONV rule is sound, even if the conversion path uses ill-typed terms. If this is the case, we can find another path only made of well-typed terms.

Corollary 2.1.11 (Using CONV is always sound)**.** *Any use of* CONV *can be broken into a sequence of reduction steps followed by a sequence of expansion steps between well-typed terms only.*

Proof. Let us suppose we have $\Gamma \vdash M : T$, $\Gamma \vdash T' : s$ and $T =_\beta T'$. By *Confluence*, there is T_0 such that $T \twoheadrightarrow_\beta T_0 \;{}_\beta\!\twoheadleftarrow T'$. By *Type Correctness*, there is t such that $\Gamma \vdash T : t$, or $T \equiv t$:

1. In the first case, by *Subject Reduction*, we know that any term that appears in the reduction from T to T_0 is typed by t, and any term that appears in the reduction from T' to T_0 is typed by s. So we have a path from T to T' exactly made of well-typed terms.

2. In the second case, $T' =_\beta t$ and by *Confluence*, $T' \twoheadrightarrow_\beta t$. *Subject Reduction* enforces $\Gamma \vdash t : s$. So this time also, the path from $T(\equiv t)$ and T' is exactly made of well-typed terms.

□

It is here interesting to see that in the first case, the path between T and T' is well-typed by sorts, but nothing guarantees that we can have the same sort in both branches. If we wanted to do so, we would need to be in a functional PTS.

About the shape of types in PTSs

Until now, we have seen some useful properties to deal with terms (*Generation, Subject Reduction, ...*) but almost nothing to deal with types (*Type Correctness* and, for functional PTSs only, *Type Uniqueness*). In [vBJ93], Jutting made a deeper study of the types in PTSs to express a more general

property than *Type Uniqueness*. He started by splitting the terms in two distinct families:

Terms Classifaction in PTSs There is a partition of terms in two sets Tv and Ts:

- for all $x \in Vars$, $x \in Tv$.
- for all $M \in Tv$, for all A, N, $\lambda x^A.M \in Tv$ and $M\ N \in Tv$.
- for all $s \in Sorts$, $s \in Ts$.
- for all A, B, $\Pi x^A.B \in Ts$.
- for all $M \in Ts$, for all A, N, $\lambda x^A.M \in Ts$ and $M\ N \in Ts$.

The class Tv can be seen as a closure for the variables by abstraction and application, and Ts as the same closure for sorts and Π-types. The former is "informally" the set of *values*, which contains variables and functions, and the latter is "informally" the set of *types*, which contains sorts and function types. This is not totally true but it is the rough idea behind this classification.

Both families enjoy a very particular property about their types. But we first need to define the notion of *telescope*

Telescopes A Π-telescope $\Pi\Gamma.B$ (resp. λ-telescope $\lambda\Gamma.M$) is defined as :

- $\Pi\emptyset.B \triangleq B$ (resp. $\lambda\emptyset.M \triangleq M$).
- $\Pi(x : A)\Gamma.B \triangleq \Pi x^A \Pi\Gamma.B$ (resp. $\lambda(x : A)\Gamma.M \triangleq \lambda x^A \lambda\Gamma M$).

Theorem 2.1.12 (The Shape of Types)**.**
If $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *then:*

- *either* $M \in Tv$ *and* $A =_\beta B$
- *or* $M \in Ts$ *and there are* Δ, s *and* t *such that* $A \twoheadrightarrow_\beta \Pi\Delta.s$ *and* $B \twoheadrightarrow_\beta \Pi\Delta.t$

Proof. The proof looks very similar to the proof of *Type Uniqueness*. For the first item, we just avoid the two problematic cases SORT and PI since sorts and Π-types are not in Tv. The second item may seem complicated but the proof is straightforward thanks to the *Confluence* of β-reduction. □

We can see now where the *Uniqueness of Types* comes from: with the functional hypothesis, both sorts in the second case will always be equal, and so in both cases, we have $A =_\beta B$. However, in the general case, the two types are telescopes which only differ by their very last sort.

There is a last property we need to present, but this time only for the set Ts. We already have information on the type of a term in Ts, but what about its shape ?

Lemma 2.1.13 (Shape of terms in Ts).
If $M \in Ts$, $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then there are Δ, K, s, t such that:

- $M \twoheadrightarrow_\beta \lambda\Delta.K$, $A \twoheadrightarrow_\beta \Pi\Delta.s$ *and* $B \twoheadrightarrow_\beta \Pi\Delta.t$.
- $\Gamma \vdash \lambda\Delta.K : A$ *and* $\Gamma \vdash \lambda\Delta.K : B$.

where K is a sort or a Π-type.

The purpose of this lemma is to exhibit the heart of a term in Ts. After enough applications to get rid of the leading λ-abstraction, we *always* find a sort or a Π-type, and the λ-telescope inside M is syntactically the same than the Π-telescope inside its type.

These notions of *Shape of Types* and *Shape of Terms* are at the core of the proof of *Strengthening* for PTSs by Jutting. This is a very technical proof and we will not need it for our developments, so we will not detail it. However, these properties on the shape of terms will be central to another crucial proof in the next chapter.

2.2 Pure Type Systems in Sequent Calculus

As we have just seen, Pure Type Systems have been studied a lot through the lens of natural deduction, but some attempts have been done in the direction of sequent calculus, like the works on *Cut Elimination* or *Expansion Postponement* by Gutierrez and Ruiz [GR02, GR03].

During our investigation on the meta-theory of PTSs, we also faced the latter. *Expansion Postponement* [Pol92, Fan97] is a problem raised by Barendregt and Pollack in the early 90's. It arise from the fact that, during the typing, doing only reductions seems most of the time enough (especially for

the application case). So the intuitive idea is to try to postpone all the expansion steps at the end of the derivation. The idea started from Pollack's work on finding a suitable type checking algorithm for Pure Type Systems [Pol92] by splitting the conversion rule and replace it by two separate rules, one to do reduction and one to do expansion:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash M : B}\ \textsc{conv}$$

$$\Updownarrow$$

$$\frac{\Gamma \vdash M : A \qquad A \to_\beta B \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ \textsc{red} \qquad \frac{\Gamma \vdash M : A \qquad B \to_\beta A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ \textsc{exp}$$

With this presentation, he tried to get rid of the EXP rule by pushing it at the end of the derivations, hence the name *Expansion Postponement*. Such a transformation of the type system allow to simply rely on reduction, to get rid of the expansion steps and thus ensure that we do not have to guess any type involved in the conversion process. *Expansion Postponement* has been proved to be a necessary condition to be able to effectively type check a Pure Type Systems, but not a sufficient one. Some partial results (see [vBJMP93, GR03]), and especially the case of *normalizing* PTSs [Pol98] have been proved correct, but the general question stays open.

Unfortunately, we do not give an answer to this question, but a new approach and a new framework to the problem, which may lead to new ideas in this field.

We describe in this section a presentation of Pure Type Systems based on the $\overline{\lambda}$-sequent calculus of Herbelin [Her94, Her95], which serves as a formalization of Lengrand's sequent-calculus [Len06] inside *Coq* [Silb]. Lengrand already proved that both presentations are completely equivalent by presenting a translation from natural deduction into sequent calculus, and back. But they are not really designed for the same purpose: As a natural deduction system, PTSs really look like a programming language, but as a sequent calculus, they seem more designed to do efficient proof-search, type inference or design abstract machines, mainly because of the way applications are handled, with explicit lists of arguments instead of the usual "unary" application of natural deduction.

2.2.1 Terms and Reduction

The main difference between terms used in natural deduction and those used in the $\overline{\lambda}$ of Herbelin is the explicit use of terms and *lists of terms* to build the applications. Also, variables do not live by themselves, they are always applied to such a list. Doing so, we can very easily define the set of normal terms by removing the general application $(M\ l)$ constructor from the terms definition.

Structure of terms and contexts
$s\ :\ Sorts$
$x\ :\ Vars$
$A, B, M\ ::=\ s \mid (x\ l) \mid (M\ l) \mid \lambda x^A.M \mid \Pi x^A.B$
$l\ ::=\ \bullet \mid M :: l$

$\Gamma\ ::=\ \emptyset \mid \Gamma(x : A)$

The basic definitions about terms are the same as for natural deduction: contexts are defined in the same way, variables and bindings behave the same, and the set of *Sorts* is still a parameter of these PTSs. However, we introduce a new syntactic family of terms: lists of terms. They are also known as "spines" (see [CP03] for a formal definition of the *Spine Calculus*, but one has just to recall that spines and lists of terms are the very same thing). In natural deduction, when a function has more then one argument, and is fully applied, we end up having a term like $(((f\ x)\ y)\ z)$. Extracting f from such a term can be costly: one needs to parse the term until the last application. In $\overline{\lambda}$, the head of an application is directly available: the previous term can be encoded as $(f\ x :: y :: z :: \bullet)$. This syntax is quite useful when doing proof search, unification, or in any situation where the head of an application need to be checked before its arguments, e.g. to known how many arguments needs to be applied, or to get the resulting type.

As for the syntax, the empty spine is denoted by $\bullet$, and the concatenation of two spines l_1 and l_2 is written $l_1 @ l_2$.

Lengrand and Herbelin added explicit substitutions to this definition, in order to make the proof of *Strong Normalization* easier. Since we are not interested in this property right now, it seems normal to reduce the number of rules and to use an an external notion of substitution. We differ from

Lengrand's thesis on this point and use the same substitution as before, with its natural extension to lists:

1. $\bullet[N/x] := \bullet$
2. $(M :: l)[N/x] := M[N/x] :: l[N/x]$

However, we will see in the next section that all the typing rules that deal with substitutions in Lengrand's work can be proved in our framework. The usual notion of β-reduction also needs to be adapted to the concept of lists:

β-reduction in Sequent Calculus β-reduction is defined by the main reduction steps

- $(\lambda x^A.M\ [N :: l]) \ \to_\beta\ (M[N/x]\ l)$
- $(M\ \bullet)\ \to_\beta\ M$
- $(M\ l_1)\ l_2\ \to_\beta\ (M\ l_1@l_2)$
- $(x\ l_1)\ l_2\ \to_\beta\ (x\ l_1@l_2)$

closed by congruence on our syntax.

The reduction in contexts $\Gamma \to_\beta \Gamma'$ is defined in the same way as for natural deduction.

This reduction also enjoys the *Confluence* and *Church-Rosser* properties, as shown in [Len06]. The proof relies on a translation of the system in the natural deduction framework where the property is well-known. A direct proof of those theorems can also be done by defining a parallel β-reduction for $\overline{\lambda}$, and use a variant of the classic approach with parallel reduction adapted to sequent calculus.

2.2.2 Confluence of β-reduction in $\overline{\lambda}$

The proof of β-confluence in natural deduction is a well-known property, which can be easily achieved by building a "parallel" reduction [Bar84] in the sense that it can reduce several independent redexes at the same time. Lengrand proved it by simulating the β-reduction of sequent calculus through the natural deduction's version. A more direct approach, inspired by the parallel reduction can be done directly in sequent calculus, but we need to be really careful about the flattening rules for lists of terms:

$$(M\ l_1)\ l_2\ \rightarrow_\beta\ (M\ l_1@l_2) \text{ and } (x\ l_1)\ l_2\ \rightarrow_\beta\ (x\ l_1@l_2).$$

If we allow too much power (or too few) to the parallel counterpart of these rules, it can easily make the confluence property unprovable.

The main idea of building a parallel reduction is to build a reduction that enjoys the diamond property:

Diamond Property if $M \rightarrow_{\beta_{//}} N$ and $M \rightarrow_{\beta_{//}} P$, then there is a Q such that $N \rightarrow_{\beta_{//}} Q$ and $P \rightarrow_{\beta_{//}} Q$.

Then we only need a few closure lemmas to show that if the parallel version enjoys the diamond property, then β-reduction enjoys the confluence property.

We propose in Fig. 2.2 a simple presentation of the parallel β-reduction in sequent calculus, where we are using the following function:

Definition The $flat$ function

$$\begin{array}{lcl} flat((x\ l_1), l_2) & \triangleq & (x\ l_1@l_2) \\ flat((M\ l_1), l_2) & \triangleq & (M\ l_1@l_2) \\ flat(M, \bullet) & \triangleq & M \\ flat(M, N :: l) & \triangleq & (M\ N :: l) \end{array}$$

Its main purpose is to remove the applications involving empty lists, along with flattening the lists when the head of an application reduces itself to an application. We need some properties on $flat$ to achieve the diamond property of the parallel reduction:

Lemma 2.2.1 (Properties of the $flat$ function)**.**

1. *$flat\ (flat\ M\ l_1)\ l_2\ =\ flat\ M\ l_1@l_2$.*
2. *$(M\ l) \twoheadrightarrow_\beta flat\ M\ l$.*
3. *If $M \rightarrow_{\beta_{//}} N$ and $l_1 \rightarrow_{\beta_{//}} l_2$, then $flat\ M\ l_1 \rightarrow_{\beta_{//}}\ flat\ N\ l_2$.*
4. *If $M \twoheadrightarrow_\beta N$ and $l_2 \twoheadrightarrow_\beta l_2$, then $flat\ M\ l_1 \twoheadrightarrow_\beta\ flat\ N\ l_2$.*

Proof. The proofs of 1 and 2 are straightforward by induction on M. The proofs of 3 and 4 are done by induction on the first reduction but need an additional property:

$$\frac{M_1 \rightarrow_{\beta_{//}} M_2 \qquad l_1 \rightarrow_{\beta_{//}} l_2}{(M_1\ l_1) \rightarrow_{\beta_{//}} (M_2\ l_2)} \qquad \frac{l_1 \rightarrow_{\beta_{//}} l_2}{(x\ l_1) \rightarrow_{\beta_{//}} (x\ l_2)}$$

$$\frac{A_1 \rightarrow_{\beta_{//}} A_2 \qquad M_1 \rightarrow_{\beta_{//}} M_2}{\lambda x^{A_1}.M_1 \rightarrow_{\beta_{//}} \lambda x^{A_2}.M_2} \qquad \frac{A_1 \rightarrow_{\beta_{//}} A_2 \qquad B_1 \rightarrow_{\beta_{//}} B_2}{\Pi x^{A_1}.B_1 \rightarrow_{\beta_{//}} \Pi x^{A_2}.B_2}$$

$$\frac{M_1 \rightarrow_{\beta_{//}} M_2 \qquad l_1 \rightarrow_{\beta_{//}} l_2}{(M_1\ l_1) \rightarrow_{\beta_{//}} \texttt{flat}\ M_2\ l_2} \qquad \frac{M_1 \rightarrow_{\beta_{//}} M_2 \qquad l_1 \rightarrow_{\beta_{//}} l_2}{M_1 :: l_1 \rightarrow_{\beta_{//}} M_2 :: l_2}$$

$$\frac{}{\bullet \rightarrow_{\beta_{//}} \bullet} \qquad \frac{}{s \rightarrow_{\beta_{//}} s}$$

$$\frac{\begin{array}{c} M_1 \rightarrow_{\beta_{//}} M_2 \\ N_1 \rightarrow_{\beta_{//}} N_2 \qquad l_1 \rightarrow_{\beta_{//}} l_2 \end{array}}{(\lambda x^A.M)\ (N :: l) \rightarrow_{\beta_{//}} (M_2[N_2/x]\ l_2)}$$

Figure 2.2: Parallel β-reduction

1. If $l_1 \twoheadrightarrow_\beta l_2$ and $k_1 \twoheadrightarrow_\beta k_2$ then $l_1 @ k_2 \twoheadrightarrow_\beta l_2 @ k_2$ holds.
2. If $l_1 \rightarrow_{\beta_{//}} l_2$ and $k_1 \rightarrow_{\beta_{//}} k_2$ then $l_1 @ k_2 \rightarrow_{\beta_{//}} l_2 @ k_2$ holds.

□

All these properties are the key point to prove the *Diamond Property* of the parallel reduction:

Proof. The proof follows Takahashi's approach [Tak95] except that we need to use the correct properties of the *flat* function to close the cases involving flattening. The main difficulty was to design the correct *flat* function, then the proof is mainly induction. □

If the one-step parallel reduction enjoys the *Diamond Property*, showing that the multi-step reduction is confluent directly follows.

Since the usual reduction and the parallel one have the same transitive closure, we directly proved that β-reduction in sequent calculus is confluent without translating to natural deduction and back.

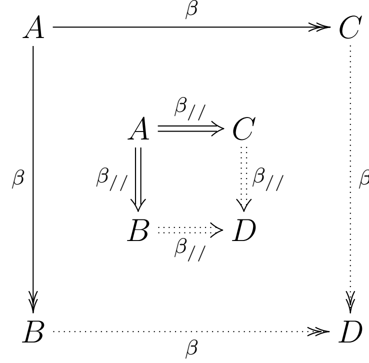

Figure 2.3: Confluence Diagram

Now that we have stated the basic properties of the terms, we can define the typing rules of sequent-calculus Pure Type Systems.

2.2.3 Typing Rules

The typing rules for sequent calculus PTSs are shown in Fig. 2.4. The system is defined by mutual definitions of well formation of contexts, terms and lists of terms.

The new judgment $\Gamma; A \vdash l : B$ defines how we type the new class of lists of terms. Lists of terms are like a bridge between a function and its result: knowing the type A of the function to which a list will be applied (A is called the *stoup*) and the final resulting type B one wants to reach (that is after applying all the terms of the list), the list l is built so that, for any term M of type A, $M\ l$ is of type B.

2.2.4 Properties of the system

Following Lengrand's thesis, we can derive the following theorems and rules.

Theorem 2.2.2 (Validity of Contexts)**.**
If $\Gamma \vdash M : T$ *or* $\Gamma; A \vdash l : B$ *then* Γ_{wf}.

Theorem 2.2.3 (Weakening)**.**

1. *If* $\Gamma_1\Gamma_2 \vdash M : B$*,* $\Gamma_1 \vdash A : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2 \vdash M : B$.

2. *If* $\Gamma_1\Gamma_2; C \vdash l : B$, $\Gamma_1 \vdash A : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2; C \vdash l : B$.

3. *If* $\Gamma_1\Gamma_2\ _{wf}$, $\Gamma_1 \vdash A : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2\ _{wf}$.

Theorem 2.2.4 (Substitution)**.**

1. *If* $\Gamma_1(x : A)\Gamma_2 \vdash M : B$ *and* $\Gamma_1 \vdash P : A$ *then* $\Gamma_1\Gamma_2[P/x] \vdash M[P/x] : B[P/x]$.

2. *If* $\Gamma_1(x : A)\Gamma_2; C \vdash l : B$ *and* $\Gamma_1 \vdash P : A$ *then* $\Gamma_1\Gamma_2[P/x]; C[P/x] \vdash l[P/x] : B[P/x]$.

$$\frac{}{\emptyset_{wf}}\ \textsc{empty} \qquad\qquad \frac{\Gamma \vdash A : s \qquad x \notin Dom(\Gamma)}{\Gamma(x : A)_{wf}}\ \textsc{extend}$$

$$\frac{\Gamma_{wf} \qquad (s,t) \in \mathcal{A}x}{\Gamma \vdash s : t}\ \textsc{sorted} \qquad\qquad \frac{\Gamma; A \vdash l : B \qquad \Gamma(x) = A}{\Gamma \vdash (x\ l) : B}\ \textsc{select}_x$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma(x : A) \vdash B : s_2 \qquad (s_1,s_2,s_3) \in \mathcal{R}el}{\Gamma \vdash \Pi x^A.B : s_3}\ \Pi wf \qquad\qquad \frac{\Gamma \vdash \Pi x^A.B : s \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.M : \Pi x^A.B}\ \Pi r$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma; A \vdash l : B}{\Gamma \vdash (M\ l) : B}\ \textsc{cut} \qquad\qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A =_\beta B}{\Gamma \vdash M : B}\ \textsc{conv}_r$$

$$\frac{\Gamma \vdash A : s}{\Gamma; A \vdash \bullet : A}\ \textsc{ax} \qquad\qquad \frac{\Gamma \vdash M : A \quad \Gamma; B[M/x] \vdash l : C \qquad \Gamma \vdash \Pi x^A.B : s}{\Gamma; \Pi x^A.B \vdash M :: l : C}\ \Pi l$$

$$\frac{\Gamma; A \vdash l : B \qquad \Gamma \vdash C : s \qquad A =_\beta C}{\Gamma; C \vdash l : B}\ \textsc{conv}_l \qquad\qquad \frac{\Gamma; A \vdash l : B \qquad \Gamma \vdash C : s \qquad B =_\beta C}{\Gamma; A \vdash l : C}\ \textsc{conv'}_r$$

Figure 2.4: Typing Rules of SC-PTSs

3. *If* $\Gamma_1(x:A)\Gamma_2\ _{wf}$ *and* $\Gamma_1 \vdash P : A$ *then* $\Gamma_1\Gamma_2[P/x]_{wf}$.

Theorem 2.2.5 (Type and Stoup Correctness)**.**

1. *If* $\Gamma \vdash M : A$ *then* $A \equiv s$ *or* $\Gamma \vdash A : s$ *for some* $s \in Sorts$.

2. *If* $\Gamma; A \vdash l : B$ *then* $\Gamma \vdash A : s$ *and* $\Gamma \vdash B : s$ *for some* $s, t \in Sorts$.

Lemma 2.2.6 (List Concatenation)**.**

If $\Gamma; A \vdash l_1 : B$ *and* $\Gamma; B \vdash l_2 : C$ *then* $\Gamma; A \vdash l_1 @ l_2 : C$.

The following *Generation* lemma gives the same amount of information than the one we already proved for PTSs in natural deduction. However, since sequent calculus involves more typing rules, leading to much more sub-derivations, we will give here a shorter statement, inspired by Lengrand's work.

Derived without conversion
We write $\Gamma \vdash^* M : A$ (resp. $\Gamma; A \vdash^* l : B$) whenever we can derive $\Gamma \vdash M : A$ (resp. $\Gamma; A \vdash l : B$) and the last rule is not a conversion rule.

Since the last rule of a judgment in $\vdash^*$ can not end by a conversion rule, we can directly fetch the typing information of the sub-terms directly by looking at the last rule used in the derivation:

Lemma 2.2.7 (Generation Lemma)**.**

1. (a) *If* $\Gamma \vdash s : C$ *then there is* t *such that* $\Gamma \vdash^* s : t$ *and* $C =_\beta t$.
   (b) *If* $\Gamma \vdash \Pi x^A.B : C$ *then there is* s *such that* $\Gamma \vdash^* \Pi x^A.B : s$ *and* $C =_\beta s$.
   (c) *If* $\Gamma \vdash \lambda x^A.M : C$ *then there is* B *such that* $\Gamma \vdash^* \lambda x^A.M : \Pi x^A.B$ *and* $C =_\beta \Pi x^A.B$.
   (d) *If* M *is not of the above forms and* $\Gamma \vdash M : C$, *then* $\Gamma \vdash^* M : C$.

2. (a) *If* $\Gamma; A \vdash \bullet : B$ *then* $A =_\beta B$.
   (b) *If* $\Gamma; A \vdash M :: l : B$ *then there are* C, D *such that* $A =_\beta \Pi x^C.D$ *and* $\Gamma; \Pi x^C.D \vdash^* M :: l : B$.

We could use the translation back and forth to natural deduction to prove the main properties of this presentation, but in fact there is no need to do so, direct proofs are as easy as in the previous case. *Subject Reduction* is a good example of this point, since it is done by a simple induction:

Theorem 2.2.8 (Subject and Context Reduction)**.**

- *If* $\Gamma \vdash M : A$ *then*
  - $M \rightarrow_\beta N$ *implies* $\Gamma \vdash N : A$
  - *and* $\Gamma \rightarrow_\beta \Delta$ *implies* $\Delta \vdash M : A$.
- *If* $\Gamma; A \vdash l : B$ *then*
  - $l \rightarrow_\beta k$ *implies* $\Gamma; A \vdash k : B$
  - $\Gamma \rightarrow_\beta \Delta$ *implies* $\Delta; A \vdash l : B$.
- *If* Γ_{wf} *and* $\Gamma \rightarrow_\beta \Delta$ *then* Δ_{wf}.

Proof. By mutual induction on lists, contexts and terms, we prove each time that statements over term reduction and context reduction hold at the same time. □

Now that we have the basis for a sequent calculus PTSs, we can begin to study the different ways to use the conversion, so that we can try to understand its use and start to have control over it.

2.3 Delayed Pure Type System in Sequent Calculus

In order to study the behavior of conversion, we need more atomic rules so we can try to split them into reductions and expansions. For example, the usual approach to *Expansion Postponement* is to embed the expansions in the premises of the other rules. But then one gets stuck with the hypothesis of the Πr rules which resists postponement. In order to explore a new approach to the question of postponement, we choose here an other solution: embed the reduction in some hypothesis such that only an expansion rule remains.

It introduces a kind of "delay" in the typing of the hypothesis as we do not check the well-formation of types, but rather the well-formation of an

expanded version of those types. The motivation behind this choice is to let some freedom to the reduction to change the shape of a type, but we still need some guaranties that the types produced are well-formed. When this strategy will have been proved valid by having the *Subject Reduction* property, we will focus on testing whether we can postpone expansions in this system.

2.3.1 Typing Rules

The typing rules of the alternative system are shown in Fig 2.5. It is interesting to notice that we can make a variant of this system by replacing the CUT_d with this one:

$$\frac{\begin{array}{c}\Gamma \vdash_d M : A \\ \Gamma; A' \vdash_d l : B \qquad \Gamma \vdash_d A'' : s \qquad A'' \twoheadrightarrow_\beta A' \qquad A'' \twoheadrightarrow_\beta A\end{array}}{\Gamma \vdash_d (M\ l) : B}\ \text{CUT'}_d$$

Both presentations are strictly equivalent, and the use of the variant system can actually ease the proof of the *Substitution* lemma. For the sake of clarity, we kept the usual shape of the CUT rule in our system, in order to keep the delay where it is mandatory.

2.3.2 Properties of the system

In order to achieve the *Subject Reduction* property and validate our system, we only need the usual lemmas, but the order changes a little, since we no longer have the reduction rule at hand. Very few changes are made to the statements, and the proof are mostly the same.

Weakening and *Substitution* are the same, but there is now a variant of the substitution lemma which can be useful to prove *Subject Reduction* (but not mandatory). We need *Type Reduction* (see below) to prove it, but it is interesting to note that we can derive it directly in the variant system using CUT'_d.

Theorem 2.3.1 (Variant Substitution with common expand)**.**

1. *If* $\Gamma_1(x : A')\Gamma_2 \vdash_d M : B$*,* $\Gamma_1 \vdash_d P : A$*,* $\Gamma_1 \vdash_d A'' : s$*,* $A'' \twoheadrightarrow_\beta A$ *and* $A'' \twoheadrightarrow_\beta A'$*, then* $\Gamma_1\Gamma_2[P/x] \vdash_d M[P/x] : B[P/x]$*.*

$$\frac{}{\emptyset_{wf_d}}\ \text{EMPTY}_d \qquad \frac{\Gamma \vdash_d A : s \qquad x \notin Dom(\Gamma)}{\Gamma(x:A)_{wf_d}}\ \text{EXTEND}_d$$

$$\frac{\Gamma_{wf_d} \qquad (s,t) \in \mathcal{A}x}{\Gamma \vdash_d s : t}\ \text{SORTED}_d \qquad \frac{\Gamma; A \vdash_d l : B \qquad \Gamma(x) = A}{\Gamma \vdash_d (x\ l) : B}\ \text{SELECT}_{x\ d}$$

$$\frac{\Gamma \vdash_d A : s_1 \qquad \Gamma(x:A) \vdash_d B : s_2 \qquad (s_1,s_2,s_3) \in \mathcal{R}el}{\Gamma \vdash_d \Pi x^A.B : s_3}\ \Pi wf_d \qquad \frac{\Gamma \vdash_d M : A \qquad \Gamma; A \vdash_d l : B}{\Gamma \vdash_d (M\ l) : B}\ \text{CUT}_d$$

$$\frac{\Gamma \vdash_d A' : s_1 \qquad \Gamma, x : A' \vdash_d B' : s_2 \qquad \Gamma, x : A' \vdash_d M : B \qquad A' \twoheadrightarrow_\beta A \qquad B' \twoheadrightarrow_\beta B \qquad (s_1,s_2,s_3) \in \mathcal{R}el}{\Gamma \vdash_d \lambda x^{A'}.M : \Pi x^A.B}\ \Pi r_d$$

$$\frac{\Gamma \vdash_d M : A \qquad \Gamma \vdash_d B' : s \qquad B' \twoheadrightarrow_\beta B \qquad B \twoheadrightarrow_\beta A}{\Gamma \vdash_d M : B}\ \text{EXP}_d$$

$$\frac{\Gamma \vdash_d A' : s \qquad A' \twoheadrightarrow_\beta A}{\Gamma; A \vdash_d \bullet : A}\ \text{AX}_d$$

$$\frac{\Gamma \vdash_d M : A \qquad \Gamma; B[M/x] \vdash_d l : C \qquad \Gamma \vdash_d T : s \qquad T \twoheadrightarrow_\beta \Pi x^A.B}{\Gamma; \Pi x^A.B \vdash_d M :: l : C}\ \Pi l_d$$

$$\frac{\Gamma; A \vdash_d l : B \qquad \Gamma \vdash_d C' : s \qquad C' \twoheadrightarrow_\beta C \qquad C \twoheadrightarrow_\beta A}{\Gamma; C \vdash_d l : B}\ \text{EXP}_{l\ d}$$

$$\frac{\Gamma; A \vdash_d l : B \qquad \Gamma \vdash_d C' : s \qquad C' \twoheadrightarrow_\beta C \qquad C \twoheadrightarrow_\beta B}{\Gamma; A \vdash_d l : C}\ \text{EXP}_{r\ d}$$

Figure 2.5: Typing Rules for the Delayed System

2. *If* $\Gamma_1(x : A')\Gamma_2; C \vdash_d l : B$, $\Gamma_1 \vdash_d P : A$, $\Gamma_1 \vdash_d A'' : s$, $A'' \twoheadrightarrow_\beta A$ *and* $A'' \twoheadrightarrow_\beta A'$, *then* $\Gamma_1\Gamma_2[P/x]; C[P/x] \vdash_d l[P/x] : B[P/x]$.

3. *If* $\Gamma_1(x : A')\Gamma_2\ {}_{wfd}$,$\Gamma_1 \vdash_d P : A$, $\Gamma_1 \vdash_d A'' : s$, $A'' \twoheadrightarrow_\beta A$ *and* $A'' \twoheadrightarrow_\beta A'$, *then* $\Gamma_1\Gamma_2[P/x]_{wfd}$.

We cannot yet prove the *Type* and *Stoup Correction* theorems, we will need *Type Reduction* to do so.

The *Generation* Lemma is changed accordingly to our news conversion conventions:

Derived without conversion We write $\Gamma \vdash^*_d M : A$ (resp. $\Gamma; A \vdash^*_d l : B$) whenever we can derive $\Gamma \vdash_d M : A$ (resp. $\Gamma; A \vdash_d l : B$) and the last rule is not an expansion rule.

Lemma 2.3.2 (Generation lemmas)**.**

1. (a) *If* $\Gamma \vdash_d s : C$ *then there is* t *such that* $\Gamma \vdash^*_d s : t$ *and* $C \twoheadrightarrow_\beta t$.
   (b) *If* $\Gamma \vdash_d \Pi x^A.B : C$ *then there is* s *such that* $\Gamma \vdash^*_d \Pi x^A.B : s_3$ *and* $C \twoheadrightarrow_\beta s_3$.
   (c) *If* $\Gamma \vdash_d \lambda x^A.M : C$ *then there are* A', B', B'', s_A, s_B *such that*
      - $\Gamma \vdash^*_d \lambda x^A.M : \Pi x^{A'}.B'$, $\Gamma \vdash_d A : s_A$ *and* $\Gamma, x : A \vdash_d B'' : s_A$
      - $A \twoheadrightarrow_\beta A'$, $B'' \twoheadrightarrow_\beta B'$ *and* $C \twoheadrightarrow_\beta \Pi x^{A'}.B'$.
   (d) *If* u *is not of the above forms and* $\Gamma \vdash_d M : C$, *then* $\Gamma \vdash^*_d M : C$.

2. (a) *If* $\Gamma; A \vdash_d \bullet : B$ *then there are* C, A', B', s_A, s_B *such that*
      - $\Gamma \vdash_d A' : s_A$ *and* $\Gamma \vdash_d B' : s_B$
      - $A' \twoheadrightarrow_\beta A$, $B' \twoheadrightarrow_\beta B$, $A \twoheadrightarrow_\beta C$ *and* $B \twoheadrightarrow_\beta C$
   (b) *If* $\Gamma; A \vdash_d M :: l : B$ *then there are* C, D, T, s *such that*
      - $\Gamma \vdash_d T : s$ *and* $\Gamma; \Pi x^C.D \vdash^*_d M :: l : B$
      - $T \twoheadrightarrow_\beta A$ *and* $A \twoheadrightarrow_\beta \Pi x :^C .D$

Now we need to prove a multi-step *Type Reduction* lemma in order to get one-step *Subject Reduction* and the rest of the usual propositions.

Theorem 2.3.3 (Type Reduction)**.**

1. *If* $\Gamma \vdash_d M : A$ *and* $A \twoheadrightarrow_\beta A'$ *then* $\Gamma \vdash_d M : A'$.

2. *If* $\Gamma \vdash_d; A \vdash_d l : B$, $A \twoheadrightarrow_\beta A'$ *and* $B \twoheadrightarrow_\beta B'$ *then* $\Gamma; A' \vdash_d l : B'$.

Proof. By mutual induction on the typing judgment, most cases rely on the confluence properties (sorts reduce to sorts and Π-types reduce to Π-types). The proof is quite straightforward: since we removed the *reduction* part of the conversion rules, we will use the delay introduced in the other rules to simulate those reduction steps. □

This lemma is the main tool to finish proving the meta-theory of this delayed system: we removed the reduction part of the conversion in the typing system, and in fact, this property shows that it is still admissible.

Lemma 2.3.4 (List Concatenation)**.**
If $\Gamma; A \vdash_d l : B$ *and* $\Gamma; B \vdash_d l' : C$ *then* $\Gamma; A \vdash_d l@l' : C$.

Now we can prove *Subject Reduction* along with *Context Reduction.*

Theorem 2.3.5 (Subject and Context Reduction)**.**

- *If* $\Gamma \vdash_d M : A$ *then*
  - $M \to_\beta N$ *implies* $\Gamma \vdash_d N : A$.
  - $\Gamma \to_\beta \Gamma'$ *implies* $\Gamma' \vdash_d M : A$.
- *If* $\Gamma; A \vdash_d l : B$ *then*
  - $l \to_\beta k$ *implies* $\Gamma; A \vdash_d k : B$.
  - $\Gamma \to_\beta \Gamma'$ *implies* $\Gamma'; A \vdash_d l : B$.
- *If* Γ_{wf_d} *and* $\Gamma \to_\beta \Gamma'$ *then* Γ'_{wf_d}.

Proof. By mutual induction on lists, contexts and terms, we prove each time that statements over term reduction and context reduction hold at the same time. But we can no longer tweak the types of derivations with the reduction rule, that is why we needed to prove *Type Reduction* before doing this proof. □

With all this, we can finally prove *Type Correctness* and *Stoup Correction*, along with full type conversion for term, stoup and list.

Theorem 2.3.6 (Type Correctness).

- *If* $\Gamma \vdash_d M : A$ *then* $A \equiv s$ *or* $\Gamma \vdash_d A : s$ *for some* $s \in Sorts$.
- *If* $\Gamma; A \vdash_d l : B$ *then* $\Gamma \vdash_d B : s$ *for some* $s \in Sorts$.

Theorem 2.3.7 (Stoup Correction).
If $\Gamma; A \vdash_d l : B$ *then there is* $s \in Sorts$ *such that* $\Gamma \vdash_d A : s$.

Theorem 2.3.8 (Type and Stoup Conversion).

- *If* $\Gamma \vdash_d M : A$, $\Gamma \vdash_d B : s$ *and* $A =_\beta B$, *then* $\Gamma \vdash_d M : B$.
- *If* $\Gamma; A \vdash_d l : B$, $\Gamma \vdash_d C : s$ *and* $A =_\beta C$, *then* $\Gamma; C \vdash_d l : B$.
- *If* $\Gamma; A \vdash_d l : B$, $\Gamma \vdash_d C : s$ *and* $B =_\beta C$, *then* $\Gamma; A \vdash_d l : C$.

Now that we have the *Church-Rosser* and *Subject Reduction* properties in both system, we are able to prove that both systems are equivalent.

Theorem 2.3.9 (Equivalence of the Delayed SC-PTSs).

$$\begin{array}{lcl} \Gamma \vdash M : T & \iff & \Gamma \vdash_d M : T \\ \Gamma; A \vdash l : B & \iff & \Gamma; A \vdash_d l : B \\ \Gamma_{wf} & \iff & \Gamma_{wf_d} \end{array}$$

Proof. By induction, on the judgment, we use *Church-Rosser* and *Subject Reduction* to prove that delayed terms are in fact well-typed, or to split the conversion hypothesis in reduction and expansion steps. The proof is quite straightforward since we proved all kinds of conversion (in type or in stoup) lemmas for both systems. □

2.4 Expansion Postponement in Delayed System

With the system we developed in the previous section, we can achieve a weaker form of postponement: we can prove that the $\vdash_d$ system has the *Expansion Postponement* property. However, we have to be careful, even if this system is equivalent to the usual presentation, it does not mean that we managed to prove *Expansion Postponement* for the general case. One has to take into account that we still got the delay in our hypothesis, and if we

remove the expansion rule in the delayed system, we will not be equivalent to the first type system anymore. If we try to translate this proof of postponement from the $\vdash_d$ system into a proof of postponement of the $\vdash$ system, we will fail at the same point as Pollack: the type checking premises of the Πr rule.

Even if we fail at giving an answer to the postponement problem in the general case, it is still interesting to study such a system, and also to check its natural deduction counterpart. By switching from natural deduction to sequent calculus, we were able to find the properties of delay that were invisible in the natural deduction world. The system $\vdash_d$ did not have any real equivalent in natural deduction style, other than the standard presentation of PTSs. But if we delay the expansion rules, it makes us except another presentation in natural deduction where the weaker form of *Expansion Postponement* also holds.

With all this, we can try to build an almost syntax directed system to do type inference.

2.4.1 Postponement in Sequent Calculus

A nice property of this new system is that we can postpone all expansions done in types at the end of the derivation tree.

Definition Let $\vdash_{d\ ep}$ be a typing system similar to $\vdash_d$ where the rules $\textsc{exp}_d$ and $\textsc{exp}_{r\ d}$ have been removed.

Theorem 2.4.1 (Type Expansion Postponement)**.**

1. *If* $\Gamma \vdash_d M : B$*, there is a* B' *such that* $B \twoheadrightarrow_\beta B'$ *and* $\Gamma \vdash_{d\ ep} M : B'$*.*
2. *If* $\Gamma; A \vdash_d l : B$*, there is a* B' *such that* $B \twoheadrightarrow_\beta B'$ *and* $\Gamma; A \vdash_{d\ ep} l : B'$*.*

Proof. This theorem needs some little steps to make this reorganization possible, which are:

1. Proving *Type Reduction* (without *Stoup Reduction*) for the system $\vdash_{d\ ep}$.
2. Proving a modified *Stoup Reduction* for this system:

   If $\Gamma; A \vdash_{d\ ep} l : B$ and $A \twoheadrightarrow_\beta A'$ then there is a B' such that $\Gamma; A' \vdash_{d\ ep} l : B'$ and $B \twoheadrightarrow_\beta B'$.

3. Then we prove a more complex postponement theorem which insures that in the case of expansion postponement for lists, the expanded form of the type is well-typed:

   If $\Gamma \vdash_d M : T$, then there is a T' such that $T \twoheadrightarrow_\beta T'$ and $\Gamma \vdash_{d\ ep} M : T'$.

   If $\Gamma; A \vdash_d l : B$, then there is $s \in Sort$, and B', B'' such that $\Gamma \vdash_{d\ ep} B'' : s$ and $\Gamma; A \vdash_{d\ ep} l : B'$ and $B'' \twoheadrightarrow_\beta B \twoheadrightarrow_\beta B'$.

   If Γ_{wf_d} then $\Gamma_{wf_{d\ ep}}$.

With this last tool, it is trivial to prove our postponement statement. □

All the delay we had to introduce in this system are here to be able to prove the *Type* and *Stoup Reduction* theorems, which are the keys to prove the equivalence between $\vdash$ and $\vdash_d$, or the *Expansion Postponement* property in $\vdash_{d\ ep}$. Since we removed every kind of reduction in the term derivation rules and most of it in the lists derivation rules, if we remove one of those delays, *Type Reduction* will fail even in the simple case of the $\textsc{ax}_d$ typing rule. And without *Type Reduction*, trying to prove *Expansion Postponement* will break at the same spot than the standard tryouts: the Π_r typing rule is still the main issue.

2.4.2 Postponement in Natural Deduction

Now that we have identified a way to design a system that have the postponement property, we can go back to the natural deduction world and try to prove exactly the same postponement theorem for the standard presentation of the system with delayed typing on the lambda rule and just reduction instead of the conversion rule. A summary of the several equivalences we have proved so far can be found in Fig. 2.6.

$$
\begin{array}{ccccc}
\uparrow\downarrow\ \ \vdash_{sc}\uparrow\downarrow & \iff & \uparrow\ \ \vdash_{scd}\uparrow & \iff & \uparrow\ \ \vdash_{scd} + \uparrow_{postp} \\
\Updownarrow & & & & \\
\vdash_{nd}\uparrow\downarrow & & \iff & & \vdash_{ndd}\downarrow + \uparrow_{postp}
\end{array}
$$

Figure 2.6: Equivalence between SC and ND systems

See Fig. 2.7 for the complete set of rules of the delayed system in natural deduction. Since we do not have the stoup in natural deduction, we need to

switch the stoup expansion on the other side of the sequent, that is why we still need the reduction rule.

$$\frac{}{\emptyset_{wf_{d\ ep}}}\ \text{EMPTY}_{d\ ep} \qquad \frac{\Gamma \vdash_{d\ ep} A : s \qquad x \notin Dom(\Gamma)}{\Gamma(x : A)_{wf_{d\ ep}}}\ \text{EXTEND}_{d\ ep}$$

$$\frac{\Gamma_{wf_{d\ ep}} \qquad (s,t) \in \mathcal{A}x}{\Gamma \vdash_{d\ ep} s : t}\ \text{SORTED}_{d\ ep} \qquad \frac{\Gamma_{wf_{d\ ep}} \qquad \Gamma(x) = A}{\Gamma \vdash_{d\ ep} x : A}\ \text{VAR}_{d\ ep}$$

$$\frac{\Gamma \vdash_{d\ ep} A : s_1 \qquad \Gamma(x : A) \vdash_{d\ ep} B : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}el}{\Gamma \vdash_{d\ ep} \Pi x^A.B : s_3}\ \Pi wf_{d\ ep}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{d\ ep} A : s_1 \qquad \Gamma, x : A \vdash_{d\ ep} B' : s_2 \\ \Gamma, x : A \vdash_{d\ ep} M : B \qquad B' \twoheadrightarrow_\beta B \qquad (s_1, s_2, s_3) \in \mathcal{R}el\end{array}}{\Gamma \vdash_{d\ ep} \lambda x^A.M : \Pi x^A.B}\ \Pi r_d$$

$$\frac{\Gamma \vdash_{d\ ep} M : \Pi x^A.B \qquad \Gamma \vdash_{d\ ep} N : A}{\Gamma \vdash_{d\ ep} M\ N : B[N/x]}\ \text{CUT}_{d\ ep} \qquad \frac{\Gamma \vdash_{d\ ep} M : A \qquad A \twoheadrightarrow_\beta B}{\Gamma \vdash_{d\ ep} M : B}\ \text{EXP}_{d\ ep}$$

Figure 2.7: Typing rules for standard PTSs with delay

Theorem 2.4.2 (Expansion Postponement with delayed typing)**.**

1. *If* $\Gamma \vdash M : T$*, then there is* T' *such that* $\Gamma \vdash_{d\ ep} M : T'$ *and* $T \twoheadrightarrow_\beta T'$*.*

2. *If* Γ_{wf} *then* $\Gamma_{wf_{d\ ep}}$*.*

Proof. We could have done it by translating from the sequent calculus world, but the several delayed typing rules in the application context part makes this task quite complex. Since the new natural deduction system is not far from the standard one, a more direct proof is quite simple.

To do the proof directly, we need to rephrase the generation lemmas in the same way that we did in sequent calculus. With this, there are two steps to the proof:

1. a variant of the *Substitution Lemma*:

   If $\Gamma_1(x : A')\Gamma_2 \vdash_{ep} t : T, \Gamma_1 \vdash_{ep} P : A$ and $A' \twoheadrightarrow_\beta A$

   then there is T' such that $\Gamma_1\Gamma_2[P/x] \vdash_{ep} t[P/x] : T'[P/x]$ and $T \twoheadrightarrow_\beta T'$.

2. a modified version of *Subject Reduction*:

   If $\Gamma \vdash_{d\ ep} M : T$ and $M \rightarrow_\beta N$, then there is T' such that $\Gamma \vdash_{d\ ep} N : T'$ and $T \twoheadrightarrow_\beta T'$.

With those new statements, the proof of our postponement theorem becomes really simple.

□

We can notice that, in the natural deduction version, there are almost no delays anymore in the hypotheses of the typing rules since the rule $\text{EXP}_{d\ ep}$ is still in the system. But we still need to deal with the typing of B in the rule Πr_d which is the critical step toward full postponement in $\vdash$.

2.5 Sequent Calculus and Type Inference

We have now at hand a system with delay and an expansion property. Based on those two points, we are able to design a type inference algorithm. To be fully usable, we still need our PTSs to have nice properties over $\mathcal{A}x$ and $\mathcal{R}el$. For example, in the following SORTED_{ti} rule, we need a way to find all the t that fits s, or in the Πwf_{ti}, we need to compute s_3 from s_1, s_2 and $\mathcal{R}el$. However, the main skeleton can be developed without those assumptions.

To do so, we need to make a separation between "inputs" and "outputs": the input should be checked before trying to compute the output, and this leads us to make some simplification of $\vdash_{d\ ep}$. The $\vdash_{ti}$ system is shown in Fig. 2.8.

- $\Gamma \vdash M :\twoheadrightarrow_\beta B$ means that there is A such that $\Gamma \vdash M : A$ and $A \twoheadrightarrow_\beta B$.
- $\Gamma \vdash M :=_\beta B$ means that there is A such that $\Gamma \vdash M : A$ and $A =_\beta B$.

Delay in A of Πr_{ti} is still mandatory to have it equivalent (modulo postponement, see below) to $\vdash_d$. If we try to remove it and try to prove equivalence to $\vdash$, we need a *Type Reduction* theorem, which does not hold in $\vdash_{ti}$ without delaying on A.

$$\frac{(s,t) \in \mathcal{A}x}{\Gamma \vdash_{ti} s : t}\ \textsc{sorted}_{ti} \qquad \frac{\Gamma; A \vdash_{ti} l : B \qquad \Gamma(x) = A}{\Gamma \vdash_{ti} (x\ l) : B}\ \textsc{select}_{x\ ti}$$

$$\frac{\Gamma \vdash_{ti} A :\twoheadrightarrow_\beta s_1 \qquad \Gamma(x:A) \vdash_{ti} B :\twoheadrightarrow_\beta s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}el}{\Gamma \vdash_{ti} \Pi x^A.B : s_3}\ \Pi wf_{ti}$$

$$\frac{\Gamma \vdash_{ti} \Pi x^{A'}.B' : s \qquad \Gamma, x : A' \vdash_{ti} M :\twoheadrightarrow_\beta B \qquad A' \twoheadrightarrow_\beta A \qquad B' \twoheadrightarrow_\beta B}{\Gamma \vdash_{ti} \lambda x^{A'}.M : \Pi x^A.B}\ \Pi r_{ti}$$

$$\frac{proviso(A) \qquad \Gamma \vdash_{ti} M : A \qquad \Gamma; A \vdash_{ti} l : B}{\Gamma \vdash_{ti} (M\ l) : B}\ \textsc{cut}_{ti}$$

Where $proviso(A)$ check that if $A = s \in Sorts$, there is $t \in Sorts$ such that $(s,t) \in \mathcal{A}x$.

$$\frac{}{\Gamma; A \vdash_{ti} \bullet : A}\ \textsc{ax}_{ti} \qquad \frac{\Gamma \vdash_{ti} M :\twoheadrightarrow_\beta A \qquad \Gamma; B[M/x] \vdash_{ti} l : C}{\Gamma; \Pi x^A.B \vdash_{ti} M :: l : C}\ \Pi l_{ti}$$

$$\frac{\Gamma; A \vdash_{ti} l : B \qquad C \twoheadrightarrow_\beta A}{\Gamma; C \vdash_{ti} l : B}\ \textsc{exp}_{ti}$$

Figure 2.8: SC System for type inference

The main difference between our $\vdash_{ti}$ system and the one developed by Lengrand is the use of $:\twoheadrightarrow_\beta$ instead of $:=_\beta$. Since they are mainly used to check the compatibility of two terms, we can work around using *Church-Rosser* and the delay to split the conversion and just keep the reduction part in the typing rule. However, if we want to stick to Lengrand's framework, we can remove the delay over A and switch the *Completeness* theorem from $\twoheadrightarrow_\beta$ to $=_\beta$.

This system has been designed with type inference in mind, which leads to a simplification of the rules. We do not check in the premises that the contexts or the stoups are correctly build, but we rather assume that those conditions have been already checked, as we would do in a practical im-

plementation. This assumption is easily proved sound since every time we extend a context, we have already inferred all the information to check that the extension is valid. Therefore, all the premises that were used to check the validity of such “inputs” are removed from the typing rules.

By getting rid of these information, we are no longer able to prove some basic property of the system, like *Validity of Contexts*, *Type Correctness* or *Stoup Correctness*. However, as we said, we now take these information for granted, and thus our lemmas will all start by assuming the correct formation of contexts and stoups. There is only one place where such assumptions are not enough, during the checking of the CUT rule. Since the stoup A is an “output” of the type inference process, we cannot guarantees that it is always typed by a sort, since there we may face *top sorts* (2.1.2). This is why we added a *proviso* to this particular rule, which ensure that if the stoup A is a sort, it can not be a top sort.

Thanks to the $\vdash_{ti}$ system, given a term M (resp. a list l), we have a type candidate we need to check. This type is computed given an already check context Γ (resp. and a stoup A). So to prove that the type is valid, we need the soundness proof.

Theorem 2.5.1 (Soundness of the delay)**.**

1. *If* $\Gamma \vdash_{ti} M : T$ *and* Γ_{wf}*, then* $\Gamma \vdash M : T$.
2. *If* $\Gamma; A \vdash_{ti} l : B$ *and* $\Gamma \vdash A : s$*, then* $\Gamma; A \vdash l : B$.

Proof. The proof goes by mutual induction on the judgments in $\vdash_{ti}$. It relies on the property of *Subject Reduction* and *Type Correctness* and on the definition of the *proviso*. Since the derivation in $\vdash_{ti}$ do not have enough type information to build a full derivation in the basic system, we need to extract them using the usual theory of PTSs. However, in the case of the CUT_{ti} rule, even these properties are not enough, and the use of the *proviso* can not be avoided. □

Then we need to prove that, assuming that a context Γ (resp. a context Γ and a stoup A) is valid, if a term M (resp. a list l) has a type T (resp. B), then the one T' that the $\vdash_{ti}$ would have computed (resp. B') is correct in the sense that $T \twoheadrightarrow_\beta T'$ (resp $B \twoheadrightarrow_\beta B'$).

Theorem 2.5.2 (Completeness)**.**

1. *If* $\Gamma \vdash M : T$*, then there is* T' *such that* $\Gamma \vdash_{ti} M : T'$ *and* $T \twoheadrightarrow_\beta T'$.

2. *If* $\Gamma; A \vdash l : B$*, then there is* B' *such that* $\Gamma; A \vdash_{ti} l : B'$ *and* $B \twoheadrightarrow_\beta B'$.

Proof. We could have done another proof by mutual induction, but there is a better thing to do this time. This system is close to the delayed system with expansion postponed, the $\vdash_{d\ ep}$ system, so we are going to use the latter as an intermediate system to our completeness result. We already have done half of the work with Theorem 2.4.1. So proving the following property would be enough:

- If $\Gamma \vdash_{d\ ep} M : T$, then $\Gamma \vdash_{ti} M : T$.
- If $\Gamma : A \vdash_{d\ ep} l : B$, then $\Gamma; A \vdash_{ti} l : B$.

The similarities between both systems make this proof straightforward, leaving only one interesting point to prove: the *proviso*. To ensure that this *proviso* is always true, we just have to rely on the *Stoup Correctness*[2] for the $\vdash_{d\ ep}$ system, which conclude this proof. □

With $\vdash_{ti}$, we join the work of Lengrand with small differences. The main one is that he gave a full syntax directed system where we only give a partial one: we still have the $\textsc{exp}_{ti}$ rule. However, this is not a problem, since doing expansion in the stoup as to be understood as "doing reduction on the input". Therefore, we only need to apply $\textsc{conv}_{ti}$ when the input list is not empty, and the type is not Π type, but only *reduces* to a Π type.

2.6 A brief look back at syntactical Pure Type Systems

In this chapter, we saw several presentations of the PTSs with a common equality: the untyped β-conversion. This equality was *external* to the typing systems, but we had enough information about it (namely *Confluence* and *Subject Reduction*) to ensure that each equality can be translated into another that only involves well-typed terms.

[2]Which is stated in the proof of Theorem 2.4.1.

Another important point is that this equality was defined directly on *terms*, without any typing constraints.

We tried to enlighten a few critical places in during the proofs to already point out to the reader where will be the issues in the following chapter, where we are going to focus on a typed version of β-conversion which mixes typing and reduction.

Part III

Typed equality

Chapter 3

Judgmental Equality

Contents

3.1 PTSs with Judgmental Equality 63

3.1.1 Typing Rules 63

3.1.2 Subject Reduction and Equivalence 65

3.2 Basic meta-theory of the annotated system . . 68

3.2.1 Definition of PTSs with Annotated Type Reduction 69

3.2.2 General properties of PTS_{atr} 74

3.2.3 The *Church-Rosser* Property in PTS_{atr} 79

3.2.4 Consequences of the *Church-Rosser* property . . . 81

3.3 Equivalence of PTS_{atr} and PTSs 83

3.3.1 Confluence of the annotation process 83

3.3.2 Consequences of the Erased Confluence 86

3.3.3 Consequences of the equivalence 88

In this chapter, we are going to only focus our efforts toward a new description of PTSs in natural deduction. However, this time, we are going to change the conversion rule, and have a more constraint equality by putting together *typing* and *reduction*. Such an approach guarantee that each conversion step is well-typed, just by construction. These Pure Type Systems are often referred to as *Pure Type Systems with Judgmental Equality* (or PTS_e), and also as *semantic* PTSs [GW94] (as opposed to syntactical PTSs). This

property about the equality is mandatory during the construction of a set-theoretical model of a type system (as pointed out in [WL]): the whole process of construction requires the equality to be typed (more precisely the BETA rule) in order to be able to build the model.

The consistency of type systems is usually achieved by means of normalization, which can be proved by several techniques like Girard's *Reducibility Sets* [Gir71, Gir72], λ-sets [Alt93] or *Normalization By Evaluation* [ACD07, Abe10] to quote the most popular ones. When one wants to add some new axioms to a system, like the *Excluded Middle* or *Proof Irrelevance*, the previous techniques may not work, and one has to rely on set-theory to build a coherence model.

This is the kind of insurance we want to have about the logic behind proof assistant, to be sure we can not derive false proofs in those systems. But the actual implementation of those software mostly relies on the *untyped equality* we just presented. Therefore, the question to know whether both presentations express the same theory is crucial : can we apply the model built with a typed equality to the actual code of the proof assistant ? This is the main reason why we want to study this equality and show the equivalence between syntactical and semantic Pure Type Systems.

In this chapter, we are going to describe the whole process of the equivalence, by building an intermediate system which will be the bridge between PTSs and PTS_e. In the first part, We start by describing the meta-theory of PTS_e along with the problems involved by the new notion of equality, then we define a new system with more typing information, and explain how it can be used to prove this equivalence. Besides giving a final answer to the equivalence question, we also give a complete meta-theory for Pure Type System with typed conversion, by being able to prove *Subject Reduction* and Π-injectivity for the typed equality, even in the case of non-normalizing systems. However, the lack of direct proofs for these two lemmas is, according to us, a serious issue and the reason why we consider that PTS_e is *not* the correct framework to deal with a typed conversion. We propose here a system which is a better candidate to deal with such conversion, called *Pure Type System with Annotated Typed Reduction*.

$$\frac{}{\emptyset_{wf}}\text{ NIL} \qquad \frac{\Gamma \vdash_e A : s \qquad x \notin Dom(\Gamma)}{\Gamma(x : A)_{wf}}\text{ CONS}$$

$$\frac{\Gamma_{wf} \qquad (s,t) \in \mathcal{A}x}{\Gamma \vdash_e s : t}\text{ SORT} \qquad \frac{(s_1, s_2, s_3) \in \mathcal{R}el \quad \Gamma \vdash_e A : s_1 \qquad \Gamma(x : A) \vdash_e B : s_2}{\Gamma \vdash_e \Pi x^A.B : s_3}\text{ PI}$$

$$\frac{\Gamma_{wf} \qquad \Gamma(x) = A}{\Gamma \vdash_e x : A}\text{ VAR} \qquad \frac{\Gamma \vdash_e A : s_1 \qquad \Gamma(x : A) \vdash_e B : s_2 \quad (s_1, s_2, s_3) \in \mathcal{R}el \qquad \Gamma(x : A) \vdash_e M : B}{\Gamma \vdash_e \lambda x^A.M : \Pi x^A.B}\text{ LAM}$$

$$\frac{\Gamma \vdash_e M : A \qquad \Gamma \vdash_e A =_\beta B : s}{\Gamma \vdash_e M : B}\text{ CONV} \qquad \frac{\Gamma \vdash_e M : \Pi x^A.B \qquad \Gamma \vdash_e N : A}{\Gamma \vdash_e MN : B[N/x]}\text{ APP}$$

Figure 3.1: Typing Rules for PTS$_e$: Terms

3.1 PTSs with Judgmental Equality

3.1.1 Typing Rules

There is another variant of the presentation of Pure Type System, by defining an *internal* notion of equality: Pure Type System with Judgmental Equality, where every conversion step is checked to be well-typed.

The typing rules for PTS$_e$ are given in Fig. 3.1 and Fig. 3.2. As you can see in the two last parts of the figure, each conversion step $\Gamma \vdash _ =_\beta _ : _$ is given in detail and checked to be well-typed.

We can prove that some properties of PTSs also hold for PTS$_e$, namely *Weakening, Substitution* and *Context Conversion*. We can add to the list the following reflexivity properties (also known as *Equation Validity*) which need to be proved along with *Type Correctness*:

Lemma 3.1.1 (Type Correctness and, Left-Hand / Right-Hand reflexivity of PTS$_e$)**.**

- *If* $\Gamma \vdash_e M : T$ *or* $\Gamma \vdash_e M = N : T$*, then there is* $s \in Sorts$ *such that*

$$\frac{\Gamma \vdash_e M =_\beta N : A \qquad \Gamma \vdash_e A =_\beta B : s}{\Gamma \vdash_e M =_\beta N : B}\ \text{CONV-EQ}$$

$$\frac{\Gamma \vdash_e M =_\beta M' : \Pi x^A.B \qquad \Gamma \vdash_e N =_\beta N' : A}{\Gamma \vdash_e MN =_\beta M'N' : B[N/x]}\ \text{APP-EQ}$$

$$\frac{\Gamma_{wf} \qquad (s,t) \in \mathcal{A}x}{\Gamma \vdash_e s =_\beta s : t}\ \text{SORT-EQ} \qquad \frac{\begin{array}{c}(s_1,s_2,s_3) \in \mathcal{R}el\\ \Gamma \vdash_e A =_\beta A' : s_1 \qquad \Gamma(x:A) \vdash_e B =_\beta B' : s_2\end{array}}{\Gamma \vdash_e \Pi x^A.B =_\beta \Pi x^{A'}, B' : s_3}\ \text{PI-EQ}$$

$$\frac{\Gamma_{wf} \qquad \Gamma(x) = A}{\Gamma \vdash_e x =_\beta x : A}\ \text{VAR-EQ} \qquad \frac{\begin{array}{c}\Gamma \vdash_e A =_\beta A' : s_1 \qquad \Gamma(x:A) \vdash_e B : s_2\\ (s_1,s_2,s_3) \in \mathcal{R}el \qquad \Gamma(x:A) \vdash_e M =_\beta M' : B\end{array}}{\Gamma \vdash_e \lambda x^A.M =_\beta \lambda x^{A'}.M' : \Pi x^A.B}\ \text{LAM-EQ}$$

$$\frac{\Gamma \vdash_e M : A}{\Gamma \vdash_e M =_\beta M : A}\ \text{REFL} \qquad \frac{\begin{array}{c}(s_1,s_2,s_3) \in \mathcal{R}el\\ \Gamma \vdash_e A : s_1 \qquad \Gamma(x:A) \vdash_e B : s_2\\ \Gamma \vdash_e N : A \qquad \Gamma(x:A) \vdash_e M : B\end{array}}{\Gamma \vdash_e (\lambda x^A.M)N =_\beta M[N/x] : B[N/x]}\ \text{BETA}$$

$$\frac{\Gamma \vdash_e N =_\beta M : A}{\Gamma \vdash_e M =_\beta N : A}\ \text{SYM} \qquad \frac{\Gamma \vdash_e M =_\beta N : A \qquad \Gamma \vdash_e N =_\beta P : A}{\Gamma \vdash_e M =_\beta P : A}\ \text{TRANS}$$

Figure 3.2: Typing Rules for PTS$_e$: Equalities

$T \equiv s$ *or* $\Gamma \vdash_e T : s$.

- *If* $\Gamma \vdash_e M =_\beta N : A$*, then* $\Gamma \vdash_e M : A$.
- *If* $\Gamma \vdash_e M =_\beta N : A$*, then* $\Gamma \vdash_e N : A$.

Proof. We need to prove all these propositions at once for three main reasons:

1. to prove *Type Correctness*, we need the *Right-Hand reflexivity* for the CONV rule.

2. to prove both reflexivity statement, we need *Type Correctness* for the APP-EQ rule.

3. because of the SYM rule, we need to prove both reflexivity statement at once.

Then, *Left-Hand reflexivity* is simply done by induction: all the premises of the typing rules of PTS_e have been chosen to correctly type the left hand-side of the equality in the current context. However, the *Right-Hand reflexivity* needs a little more work: the proof relies on the *Substitution Lemma* (to type the right part of BETA), *Left Reflexivity* and *Context Conversion*.

It is interesting to notice that we could have removed the dependency on *Type Correctness* just by adding more typing information (like the fact that A and B are also well-typed, with the correct sorts) to the premises of APP-EQ. □

With these few results, we can prove half of the equivalence we are looking for:

Theorem 3.1.2 (From PTS_e to PTSs)**.**

1. *If* $\Gamma \vdash_e M : A$ *then* $\Gamma \vdash M : A$.

2. *If* $\Gamma \vdash_e M =_\beta N : A$ *then* $\Gamma \vdash M : A$, $\Gamma \vdash N : A$ *and* $M =_\beta N$.

Proof. The proof is a simple induction and relies on properties of PTSs: we just "forget" some typing information when dealing with the typed equalities. □

3.1.2 Subject Reduction and Equivalence

We previously saw that *Subject Reduction* and Π-*injectivity* were two important properties of PTSs: *Subject Reduction* allows us to freely compute without having to check that typing is preserved at every reduction step, and Π-*injectivity* is a crucial step to prove the latter. With the basic meta-theory for PTS_e at hand, we can now try to check if both properties also hold when the equality is checked to be well-typed. If it is the case, we would be able to prove that both presentations are in fact two different ways to describe the same theory.

Theorem 3.1.3 (Subject Reduction).
If $\Gamma \vdash_e M : T$ *and* $M \rightarrow_\beta N$ *then* $\Gamma \vdash_e M =_\beta N : T$.

To prove this property for PTS$_e$, we can try the same approach that was used for PTSs, but this requires to have the Π-*injectivity* for PTS$_e$. Since we are using a typed equality, we can express this injectivity in several ways, for example by completely getting rid of the types (as we did for PTSs), or instead by trying to keep as much typing information as we can.

With the first solution, we lack too much type information to build the typed equality needed by *Subject Reduction*. For the second one, we need to find the correct statement for the injectivity. After proving the equivalence between *functional* PTSs and PTS$_e$, Adams did manage to prove a strong version of injectivity, but was unsuccessful at doing it in the general case. In fact, this statement is wrong in the general case. Since we did not find any proof of this fact, we propose here a simple counter-example that prove our point:

Lemma 3.1.4 (Strong Π-injectivity does not hold for all PTS$_e$).
The following statement does not hold for all PTS$_e$:

If $\Gamma \vdash_e \Pi x^A.B =_\beta \Pi x^C.D : u$*, then* $\Gamma \vdash_e A =_\beta C : s$*,* $\Gamma(x : A) \vdash_e B =_\beta D : t$ *for some* $s, t \in Sorts$ *such that* $(s, t, u) \in Rel$.

Proof. We are going to build a counterexample by selecting the right sets for *Sorts*, *Ax* and *Rel*. Let us assume that strong injectivity (1) holds for all PTS$_e$, including the following one:

- $Sorts \equiv \{u, v, v', w, w'\}$
- $Ax \equiv \{(u, v), (u, v'), (v, w), (v', w')\}$
- $Rel \equiv \{(w, w, w), (w', w', w'), (v, v, u), (v', v', u)\}$

Let us define two terms $D1 \equiv (\lambda x^v.x)\ u \equiv id_v\ u$ and $D2 \equiv (\lambda x^{v'}.x)\ u \equiv id_{v'}\ u$. We add id_v and $id_{v'}$ in front of the sort u to put a constraint on its type, they behave like *coercion*. We can prove the following properties:

1. $\emptyset \vdash_e D1 : v$ and for all T, $\emptyset \vdash_e D1 : T$ implies $T =_\beta v$.
2. $\emptyset \vdash_e D2 : v'$ and for all T, $\emptyset \vdash_e D2 : T$ implies $T =_\beta v'$.

3. with both results and the fact that $\emptyset \vdash_e u : v$ and $\emptyset \vdash_e u : v'$, we can prove

   $\emptyset \vdash_e D1 =_\beta u : v$ and $\emptyset \vdash_e D2 =_\beta u : v'$.

4. The correct choice of rules in *Rel* leads to $\emptyset \vdash_e \Pi x^{D1}.u =_\beta \Pi x^u.u : u$ and $\emptyset \vdash_e \Pi x^u.u : u =_\beta \Pi x^{D2}.u : u$, so by applying the transitivity rule, we got $\emptyset \vdash_e \Pi x^{D1}.u =_\beta \Pi x^{D2}.u : u$.

5. Since we supposed (1), either $\emptyset \vdash_e D1 =_\beta D2 : v$ or $\emptyset \vdash_e D1 =_\beta D2 : v'$.

6. In both case, one of the reflexivity lemmas and the first two items force $v =_\beta v'$ which is impossible by *Confluence* (cf Lemma 2.1.2).

This is quite technical, but the idea of the proof is simple. We have three versions of u: $D1$ which can only be typed by v, $D2$ which can only be typed by v' and u which can have both types. It is impossible to directly link $D1$ to $D2$, but it can be done by hiding them inside the domains of Π-types. By using (1), we can extract an equality between $D1$ and $D2$ from the Π-types and expose the contradiction.

□

To directly prove *Subject Reduction*, we need to find the correct injectivity statement that will give enough typing information to build the equality, but not too much so that it is still provable in all cases. In the next sections, we will see a statement that enjoys both properties, but we are not able to prove it directly from the lemmas we have right now, so we will come back to it later.

Renouncing to prove directly the Π-injectivity we need from within PTS$_e$, one may want to translate PTS$_e$ judgments in PTS ones to use their properties, but again one is stuck: even if the translation from PTS$_e$ to PTSs is almost trivial, the only translation back from PTSs to PTS$_e$ we are aware of relies itself on *Subject Reduction in PTS$_e$*. What if we try naively to translate a PTS judgment into a PTS$_e$ one ? If we proceed by induction on the derivation tree, the most interesting case is the CONV rule:

$$\frac{A =_\beta B \qquad \Gamma \vdash M : A \qquad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ \textsc{conv}$$

By induction, we can build two new judgments : $\Gamma \vdash_e M : A$ and $\Gamma \vdash_e B : s$. We now need a way to lift the equality $A =_\beta B$ into a typed equality in PTS$_e$: having *Subject Reduction* for PTS$_e$ would be sufficient. By *Confluence*, there is C such that $A \twoheadrightarrow_\beta C$ and $B \twoheadrightarrow_\beta C$, so by *Subject Reduction*, we would be able to prove that $\Gamma \vdash_e B =_\beta C : s$ and $\Gamma \vdash_e A =_\beta C : t$. It is now easy to conclude by applying CONV and SYM:

$$\dfrac{\dfrac{\Gamma \vdash_e M : A \qquad \Gamma \vdash_e A =_\beta C : t}{\Gamma \vdash_e M : C} \qquad \Gamma \vdash_e B =_\beta C : s}{\Gamma \vdash_e M : B}$$

Unfortunately for us, we still do not have enough available material at this point to prove *Subject Reduction* for PTS$_e$. We will come back to this proof after achieving the equivalence between PTSs and PTS$_e$. In the next sections, we explain our approach to prove the general equivalence, mostly influenced by Adam's TPOSR system. However, even if our new system is very similar to TPOSR, the ways to build its meta-theory have major differences.

3.2 Basic meta-theory of the annotated system

On one side, we have PTSs which only use a untyped equality, but enjoy the *Subject Reduction* property, and on the other side, we have PTS$_e$ which use a more constraint equality which is built on top of the typing judgments, but where we do not known how to prove *Subject Reduction*. In the middle, our question: are both systems the same one ?

The real problem is that we do not have enough information about the equality to keep track of the Π-types while performing a typed conversion. Since injectivity of Π-types is usually achieved by first proving *Confluence*, we can try to think about what the missing pieces of information we could add inside our typing judgments.

We have to be really careful here: if we add annotations somewhere inside the terms or the judgments, we will have to prove that the usual terms and judgments of PTSs/PTS$_e$ can be annotated into this new system. We can already try to guess what kind of property this new annotated have to verify:

let us consider a hypothetical typing system $\Vdash$ which is built on top of some modified terms that we will write M^*. We are trying to build a translation from a judgment $\Gamma \vdash M : T$ into some judgment $\Gamma^* \Vdash M^* : T^*$. Several typing rules of PTSs involve multiple premises, like the PI rule:

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

Trying to build the translation by induction on the derivation tree seems reasonable, so we would end up having two new derivations: $\Gamma_1^* \Vdash A_1^* : s_1^*$ and $\Gamma_2^*(x : A_2^*) \Vdash B_2^* : s_2^*$. We do not have yet chosen what the system $\Vdash$ will be, but we are clearly going to need a way to show that A_1^* and A_2^* are somehow related.

We also have to remember what happened when we tried to directly translate a PTS derivation into PTS_e: we needed the latter system to verify *Subject Reduction*, so our $\Vdash$ candidate will certainly have to enjoy it.

Having both problems in mind, we can now take a look at the solution we found.

3.2.1 Definition of PTSs with Annotated Type Reduction

Let us go back to the question of lifting a typing judgment from PTSs to PTS_e. To do so, we need to be able to lift a conversion $A =_\beta B$ into a typed equality judgment $\Gamma \vdash_e A =_\beta B : s$ and as said above, we would like to have *Subject Reduction* for PTS_e which itself requires the injectivity of Π-types.

A first proof of equivalence between PTSs and PTS_e has been made by Adams [Ada06] for the subclass of *functional* PTSs, a result that we later extended to the subclasses of *semi-full* and *full* PTSs [SH10]. As expected, the key step of these proofs is to build an intermediate system with two major properties:

1. It has to be equivalent to both PTSs and PTS_e.

2. It has to verify the *Church-Rosser* property.

With such a system, we can prove that it enjoys Π-*injectivity* and *Subject Reduction*, and finally translate both properties into PTS_e.

This injectivity is a direct conclusion of the *Church-Rosser* property. But since we are dealing with a typed equality, we need to build a typed version of this property. The usual way to prove it for β-reduction is to define a parallel reduction that enjoys the *Diamond Property*, and whose transitive-closure is the same closure as β-reduction. So Adams defined a typed version of this parallel reduction called *Type Parallel One Step Reduction* (TPOSR from now on) to prove his result. However, the proof of the *Church-Rosser* property for TPOSR is not so trivial to do: as we will see in more details later, additional typing information are required to conclude the proof. Adams decided to annotate applications by their co-domain, and to restrict to functional PTSs so his system would also enjoy the *Uniqueness of Types*. We used the same annotation system to show that the *Church-Rosser* property also holds for semi-full and full systems. However, to be able to prove the *Church-Rosser* property in the general framework, this was not enough.

To overcome this limitation to restricted versions of PTSs, we extended Adams' system by adding a second annotation to the applications. In his paper, he rejected this solution because it introduces a new constraint one has to check when one wants to reduce a β-redex, and he did not investigate how to handle this additional complication. Such methods have already been tried to prove normalization results for PTSs in [MW97] and for correctness and completeness results in [Str91], but we had to adapt it without any normalization requirement.

All of this has led us to define a variant of TPOSR that we call *Pure Type System based on Annotated Typed Reduction* (or PTS$_{atr}$ for short), which is the main contribution of this work. This system is built on a trade-off : this additional annotation allows us to get more information from our typing judgments, but it adds new constraints in the typed reduction that we will have to face. We will now see in details how it is defined and what are the difficulties introduced by this new annotation.

Structure of Annotated Terms

$A, B, M, N ::= \ s \mid x \mid M_{\Pi x^A.B} N \mid \lambda x^A.M \mid \Pi x^A.B$

All the other notions (context, substitution and untyped reduction) described for the terms of PTSs are defined in the same way for PTS$_{atr}$, with their natural adaptation to the annotated applications. The notation for

untyped reductions in PTS_{atr} are the same as before, with $\to_p$ for untyped parallel reduction over annotated terms, and $\twoheadrightarrow$ for its transitive closure (since PTS_{atr} is a parallel system, using a one-step parallel reduction will be easier, but its closure is still the same as the usual one-step β-reduction). We define an erasure procedure $|\ |$ by induction on the structure of terms that maps annotated PTS_{atr} terms to non-annotated ones, by recursively removing the additional typing information within the applications:

$$
\begin{array}{lcl}
|s| & \triangleq & s \\
|x| & \triangleq & x \\
|M_{\Pi x^A.B} N| & \triangleq & |M|\ |N| \\
|\lambda x^A.M| & \triangleq & \lambda x^{|A|}.|M| \\
|\Pi x^A.B| & \triangleq & \Pi x^{|A|}.|B|
\end{array}
$$

The typing rules of PTS_{atr} are presented in Fig. 3.3 and Fig. 3.4. As a shortcut, we will use the notations $\Gamma \vdash M \rhd N : A, B$ for "$\Gamma \vdash M \rhd N : A$ and $\Gamma \vdash M \rhd N : B$", and $\Gamma \vdash M \rhd ? : A$ for "there is some N such that $\Gamma \vdash M \rhd N : A$".

The transitive-closure of $\rhd$ is written as $\rhd^+$, and the transitive-symmetric closure of $\rhd$ as $\cong_\beta$, restricted to terms typed by sorts. We will not need a full notion of equality since this new judgment already embeds a notion of reduction. The $\cong_\beta$ judgment has to be understood as an equality at "the level of types", where we do not demand to keep the same sort at every transitivity step. We will need this to be able to state the *Generation Lemmas* correctly, since we do not have the *Uniqueness of Types* in the general case.

So far, we are juggling with a few variants of β-equality, so we will now recall all our notations as a remainder to avoid confusion:

Notation	Terms	Systems	Meaning
$M \equiv N$	all	all	syntactic (α-conversion)
$M =_\beta N$	non-annotated	PTSs	β-conversion
$\Gamma \vdash_e M =_\beta N : T$	non-annotated	PTS_e	β with typing constraints
$\Gamma \vdash M \cong_\beta N$	annotated	PTS_{atr}	β with typing constraints

Let us back off a while and take a look at the BETA rule. The corresponding rule in TPOSR is somehow simpler (remember, the annotation on TPOSR's applications is only the co-domain).

$$\frac{\Gamma_{wf} \qquad (s,t) \in \mathcal{A}x}{\Gamma \vdash s \rhd s : t}\ \textsc{sort} \qquad\qquad \frac{\Gamma_{wf} \qquad \Gamma(x) = A}{\Gamma \vdash x \rhd x : A}\ \textsc{var}$$

$$\frac{\begin{array}{c}(s_1, s_2, s_3) \in \mathcal{R}el \\ \Gamma \vdash A \rhd A' : s_1 \qquad \Gamma(x:A) \vdash B \rhd B' : s_2\end{array}}{\Gamma \vdash \Pi x^A.B \rhd \Pi x^{A'}.B' : s_3}\ \textsc{prod}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \rhd A' : s_1 \qquad (s_1, s_2, s_3) \in \mathcal{R}el \\ \Gamma(x:A) \vdash B \rhd ? : s_2 \qquad \Gamma(x:A) \vdash M \rhd M' : B\end{array}}{\Gamma \vdash \lambda x^A.M \rhd \lambda x^{A'}.M' : \Pi x^A.B}\ \textsc{lam}$$

$$\frac{\begin{array}{c}(s_1, s_2, s_3) \in \mathcal{R}el \\ \Gamma \vdash A \rhd A' : s_1 \qquad \Gamma(x:A) \vdash B \rhd B' : s_2 \\ \Gamma \vdash M \rhd M' : \Pi x^A.B \qquad \Gamma \vdash N \rhd N' : A\end{array}}{\Gamma \vdash M_{\Pi x^A.B} N \rhd M'_{\Pi x^{A'}.B'} N' : B[N/x]}\ \textsc{app}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \rhd ? : s_1 \qquad \Gamma \vdash A' \rhd ? : s_1 \\ \Gamma \vdash A_0 \rhd^+ A : s_1 \qquad \Gamma \vdash A_0 \rhd^+ A' : s_1 \qquad (s_1, s_2, s_3) \in \mathcal{R}el \\ \Gamma(x:A) \vdash B \rhd B' : s_2 \\ \Gamma(x:A) \vdash M \rhd M' : B \qquad \Gamma \vdash N \rhd N' : A\end{array}}{\Gamma \vdash (\lambda x^A.M)_{\Pi x^{A'}.B} N \rhd M'[N'/x] : B[N/x]}\ \textsc{beta}$$

$$\frac{\Gamma \vdash M \rhd N : A \qquad \Gamma \vdash A \rhd B : s}{\Gamma \vdash M \rhd N : B}\ \textsc{red}$$

$$\frac{\Gamma \vdash M \rhd N : A \qquad \Gamma \vdash B \rhd A : s}{\Gamma \vdash M \rhd N : B}\ \textsc{exp}$$

Figure 3.3: Typing Rules for the PTS_{atr} system

$$\frac{\begin{array}{c}\Gamma \vdash A \rhd A' : s_1 \qquad \Gamma(x:A) \vdash B \rhd B' : s_2 \\ \Gamma(x:A) \vdash M \rhd M' : B \qquad \Gamma \vdash N \rhd N' : A \qquad (s_1, s_2, s_3) \in \mathcal{R}el\end{array}}{\Gamma \vdash (\lambda x^A.M)_{(x)\ B} N \rhd M'[N'/x] : B[N/x]}$$

$$\frac{}{\emptyset_{wf}}\ \text{EMPTY} \qquad \frac{\Gamma \vdash A \rhd ? : s \qquad x \notin Dom(\Gamma)}{\Gamma(x : A)_{wf}}\ \text{EXTEND}$$

$$\frac{\Gamma \vdash M \rhd N : A}{\Gamma \vdash M \rhd^+ N : A}\ \text{R-INTRO} \qquad \frac{\Gamma \vdash M \rhd^+ N : A \qquad \Gamma \vdash N \rhd^+ P : A}{\Gamma \vdash M \rhd^+ P : A}\ \text{R-TRANS}$$

Figure 3.4: Typing Rules for the PTS_{atr} system

$$\frac{\Gamma \vdash A \rhd B : s}{\Gamma \vdash A \cong_\beta B}\ \text{EQ-INTRO} \qquad \frac{\Gamma \vdash B \cong_\beta A}{\Gamma \vdash A \cong_\beta B}\ \text{SYM}$$

$$\frac{\Gamma \vdash A \cong_\beta B \qquad \Gamma \vdash B \cong_\beta C}{\Gamma \vdash A \cong_\beta C}\ \text{TRANS}$$

Figure 3.5: Type Equality in PTS_{atr}

Why do we had to add this troublesome new condition on the domain ? The reason is quite simple: to prove the *Church-Rosser* property of TPOSR (and PTS_{atr}), we need additional information about the possible domains of a λ-abstraction. More precisely, from a judgment of the shape $\Gamma \vdash \lambda x^A.M \rhd \lambda x^{A'}.M' : \Pi x^C.B$, one would like to prove that $\Gamma \vdash A \cong_\beta C$, *without using the injectivity of* Π-*types*. In the functional and semi-full cases, we managed to infer this information from the available hypotheses during *Church-Rosser*. However, in the general framework, we were not able to do it anymore, so we had to add this additional annotation on the application to fix this issue.

Also, the reason why the BETA rule may seem complicated at first is because of a design choice: its meaning is to ensure that there is a conversion path from the annotation A attached to the λ-connector, to the annotation of the application A', where each step is *typed by the sort* s_1 (which is the first sort of the triple). The equality $\cong_\beta$ ensures that each step is typed by a sort, but does not guarantee that each step use the same one, so we can not use it directly. So we needed *another* kind of equality much like PTS_e

equality, that takes care to keep the same sort at every reduction step. But such a solution gave us a lot more work to do: two equalities mean two sets of rules to deal with during mutual induction, weird behavior when trying to glue them together, and in the end, the new equality was only used for the proof of *Confluence*, it becomes useless afterward since we always split it in reductions.

So we chose to have a smaller system with only one kind of equality, and to relate these annotations by means of reductions and expansions, where we can still control the type. We will come back to the details on this annotation in the proof for *Church-Rosser*

3.2.2 General properties of PTS$_{atr}$

From now on, we consider the general case of PTSs, without any restrictions: we can start to prove some properties of PTS$_{atr}$ (by mutual induction over $\triangleright$ and $\triangleright^+$ at once):

Lemma 3.2.1 (Weakening)**.**

1. *If* $\Gamma_1\Gamma_2 \vdash M \triangleright N : B$, $\Gamma_1 \vdash A \triangleright ? : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2 \vdash M \triangleright N : B$.
2. *If* $\Gamma_1\Gamma_2 \vdash M \triangleright^+ N : B$, $\Gamma_1 \vdash A \triangleright ? : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2 \vdash M \triangleright^+ N : B$.
3. *If* $\Gamma_1\Gamma_2\ _{wf}$, $\Gamma_1 \vdash A \triangleright ? : s$ *and* $x \notin Dom(\Gamma_1\Gamma_2)$ *then* $\Gamma_1(x : A)\Gamma_2\ _{wf}$.

Lemma 3.2.2 (Parallel Substitution)**.**

1. *If* $\Gamma_1(x : A)\Gamma_2 \vdash M \triangleright N : B$ *and* $\Gamma_1 \vdash P \triangleright P' : A$ *then* $\Gamma_1\Gamma_2[P/x] \vdash M[P/x] \triangleright N[P'/x] : B[P/x]$.
2. *If* $\Gamma_1(x : A)\Gamma_2 \vdash M \triangleright^+ N : B$ *and* $\Gamma_1 \vdash P \triangleright P' : A$ *then* $\Gamma_1\Gamma_2[P/x] \vdash M[P/x] \triangleright^+ N[P'/x] : B[P/x]$.
3. *If* $\Gamma_1(x : A)\Gamma_2\ _{wf}$ *and* $\Gamma_1 \vdash P \triangleright ? : A$ *then* $\Gamma_1\Gamma_2[P/x]_{wf}$.

Lemma 3.2.3 (Validity of Contexts)**.**

1. *For all* Γ, M, N *and* T, *if* $\Gamma \vdash M \triangleright N : T$ *then* Γ_{wf}.

2. *For all* Γ, M, N *and* T*, if* $\Gamma \vdash M \rhd^+ N : T$ *then* Γ_{wf}*.*

3. *For all* Γ, A *and* B*, if* $\Gamma \vdash A \cong_\beta B$ *then* Γ_{wf}*.*

Proof. The two first points are done by mutual induction on the typing derivations, just as we already did for PTSs. The third is a simple combination of induction and the first conclusion. □

We extend the notion of equality on terms to equality on contexts, which are nothing but ordered lists of terms:

Context Conversion

- $\emptyset \cong_\beta \emptyset$.
- If $\Gamma \cong_\beta \Gamma'$, $\Gamma \vdash A \cong_\beta B$ and $x \notin Dom(\Gamma)$, then $\Gamma(x : A) \cong_\beta \Gamma'(x : B)$.

Lemma 3.2.4 (Conversion in Context)**.**

- *If* $\Gamma \vdash M \rhd N : A$ *and* $\Gamma \cong_\beta \Gamma'$ *then* $\Gamma' \vdash M \rhd N : A$*.*
- *If* $\Gamma \vdash M \rhd^+ N : A$ *and* $\Gamma \cong_\beta \Gamma'$ *then* $\Gamma' \vdash M \rhd^+ N : A$*.*
- *If* $\Gamma \vdash A \cong_\beta B$ *and* $\Gamma \cong_\beta \Gamma'$ *then* $\Gamma' \vdash A \cong_\beta B$*.*

Lemma 3.2.5 (Left-Hand and Right-Hand Typability)**.**

1. *If* $\Gamma \vdash M \rhd N : A$ *or* $\Gamma \vdash M \rhd^+ N : A$*, then* $\Gamma \vdash M \rhd M : A$*.*
2. *If* $\Gamma \vdash M \rhd N : A$ *or* $\Gamma \vdash M \rhd^+ N : A$*, then* $\Gamma \vdash N \rhd N : A$*.*
3. *If* $\Gamma \vdash A \cong_\beta B$*, then* $\Gamma \vdash A \rhd A : s$ *and* $\Gamma \vdash B \rhd B : t$ *for some sorts* s *and* t*.*

The following lemma is an adapted version of the *Generation Lemma* introduced for PTSs. By adding both annotations, we do not have to "guess" the domain and co-domain of an application anymore.

Lemma 3.2.6 (Generation)**.**

1. *If* $\Gamma \vdash s \rhd N : T$ *then* $N \equiv s$ *and there is* t *such that* $(s, t) \in Ax$ *and either* $T \equiv t$ *or* $\Gamma \vdash T \cong_\beta t$*.*

2. *If* $\Gamma \vdash x \rhd N : T$ *then* $N \equiv x$ *and there is* A *such that* $\Gamma(x) = A$ *and* $\Gamma \vdash T \cong_\beta A$.

3. *If* $\Gamma \vdash \Pi x^A.B \rhd N : T$ *then there are* A', B', s_1, s_2, s_3 *such that* $N \equiv \Pi x^{A'}.B'$,

   $(s_1, s_2, s_3) \in Rel$, $\Gamma \vdash A \rhd A' : s_1$, $\Gamma(x : A) \vdash B \rhd B' : s_2$ *and either* $T \equiv s_3$ *or*

   $\Gamma \vdash T \cong_\beta s_3$.

4. *If* $\Gamma \vdash \lambda x^A.M \rhd N : T$ *then there are* $A', M', B, B', s_1, s_2, s_3$ *such that* $N \equiv \lambda x^{A'}.M'$, $(s_1, s_2, s_3) \in Rel$, $\Gamma \vdash A \rhd A' : s_1$, $\Gamma(x : A) \vdash B \rhd B' : s_2$, $\Gamma(x : A) \vdash M \rhd M' : B$ *and* $\Gamma \vdash T \cong_\beta \Pi x^A.B$.

5. *If* $\Gamma \vdash P_{\Pi x^U.B} Q \rhd N : T$ *then there are* $A, A', B', Q', s_1, s_2, s_3$ *such that* $(s_1, s_2, s_3) \in Rel$,

   $\Gamma \vdash A \rhd A' : s_1$, $\Gamma(x : A) \vdash B \rhd B' : t_2$, $\Gamma \vdash Q \rhd Q' : A$, $\Gamma \vdash T \cong_\beta B[Q/x]$ *and*

   - *either (*APP *case)* $U \equiv A$, $\Gamma \vdash P \rhd P' : \Pi x^A.B$ *and* $N \equiv P'_{\Pi x^{A'}.B'} Q'$ *for some* P'
   - *or (*BETA *case)* $U \equiv A''$, $P \equiv \lambda x^A.R$, $\Gamma(x : A) \vdash R \rhd R' : B$, $N \equiv R'[Q'/x]$,
     $\Gamma \vdash A_0 \rhd^+ A'' : s_1$ *and* $\Gamma \vdash A_0 \rhd^+ A : s_1$ *for some* A_0, A'', R, R'.

One of the key point to prove the *Church-Rosser* property for β-reduction (more exactly, to prove that the usual reduction and the parallel one have the same transitive closure) is that β enjoys some nice multi-step congruence properties like:

- If $A \twoheadrightarrow_\beta B$ and $C \twoheadrightarrow_\beta D$, then $\Pi x^A.C \twoheadrightarrow_\beta \Pi x^B.D$
- If $A \twoheadrightarrow_\beta B$ and $M \twoheadrightarrow_\beta N$, then $\lambda x^A.M \twoheadrightarrow_\beta \lambda x^B.N$
- ...

However, to have the same properties in PTS_{atr}, that is with type restrictions to fulfill, those lemmas can be hard to prove, especially for the application case. By only considering the functional case, which enjoys *Type Uniqueness*, Adams got rid of this trouble and managed to prove those extensions to

TPOSR quite easily. Without this uniqueness property, we need another way to be able to find the right typing information.

To prove those multi-step congruence results for PTS_{atr}, we need to check that some terms are typed by the correct sorts (for example in the application case, we need to check that terms are typed by the triple of sorts in *Rel*). One practical case is when we know that $\Gamma \vdash A \rhd ? : s$ and $\Gamma \vdash A \rhd^+ A' : t$, but we need the latter statement typed by s. With *Type Uniqueness*, we would be able to prove that $s \equiv t$, but this is not true in the general case. What we would like to do it to keep the reduction skeleton of the second statement and use it with the types of the first judgment.

Surprisingly the key lemma to solve this problem appears to be the following:

Lemma 3.2.7 (Exchange of Types)**.** *If* $\Gamma \vdash M \rhd N : A$ *and* $\Gamma \vdash M \rhd P : B$*, then* $\Gamma \vdash M \rhd N : B$ *and* $\Gamma \vdash M \rhd P : A$*.*

Proof. By induction, there are no difficult cases since we have the co-domain annotations on the applications. □

The heart of this theorem is to keep the reduction structure of a derivation and allowing to change the type annotations inside, if we have a witness that these annotations are correct. We can directly extend this result to multi-step reduction:

Corollary 3.2.8 (Exchange of Types in multi-step reduction)**.** *If* $\Gamma \vdash M \rhd^+ N : A$ *and* $\Gamma \vdash M \rhd ? : B$*, then* $\Gamma \vdash M \rhd^+ N : B$*.*

It allows us to prove that the following transitivity rule for $\rhd^+$ is admissible:

$$\frac{\Gamma \vdash M \rhd^+ N : A \qquad \Gamma \vdash N \rhd^+ P : B}{\Gamma \vdash M \rhd^+ P : A}\ \textsc{reds-trans-alt}$$

This is the key lemma to prove our multi-step congruence lemma for PTS_{atr}:

Lemma 3.2.9 (Multi-step Congruences and Generations)**.**

- *Congruences:*

- *If* $\Gamma \vdash A \rhd^+ A' : s_1$, $\Gamma(x : A) \vdash B \rhd^+ B' : s_2$ *and* $(s_1, s_2, s_3) \in Rel$, *then*
  $\Gamma \vdash \Pi x^A.B \rhd^+ \Pi x^{A'}.B' : s_3$.
- *If* $\Gamma \vdash A \rhd^+ A' : s_1$, $\Gamma(x : A) \vdash M \rhd^+ M' : B$, $\Gamma(x : A) \vdash B \rhd ? : s_2$ *and*
  $(s_1, s_2, s_3) \in Rel$, *then* $\Gamma \vdash \lambda x^A.M \rhd^+ \lambda x^{A'}, M' : \Pi x^A.B$.
- *If* $\Gamma \vdash A \rhd^+ A' : s$, $\Gamma(x : A) \vdash B \rhd^+ B' : t$, $\Gamma \vdash M \rhd^+ M' : \Pi x^A.B$, *and*
  $\Gamma \vdash N \rhd^+ N' : A$, *then* $\Gamma \vdash M_{\Pi x^A.B} N \rhd^+ M'_{\Pi x^{A'}.B'} N' : B[N/x]$.

- *(Multi-step) Generation:*
  - *If* $\Gamma \vdash \Pi x^A.B \rhd^+ N : T$ *then there are* A', B', s_1, s_2, s_3 *such that* $(s_1, s_2, s_3) \in Rel$, $N \equiv \Pi x^{A'}.B'$, $\Gamma \vdash A \rhd^+ A' : s_1$, $\Gamma(x : A) \vdash B \rhd^+ B' : s_2$ *and* $\Gamma \vdash T \cong_\beta s_3$ *or* $T \equiv s_3$.
  - *If* $\Gamma \vdash \lambda x^A.M \rhd^+ N : T$ *then there are* A', M', B, s_1, s_2, s_3 *such that*
    $(s_1, s_2, s_3) \in Rel$, $N \equiv \lambda x^{A'}.M'$, $\Gamma \vdash A \rhd^+ A' : s_1$, $\Gamma(x : A) \vdash M \rhd^+ M' : B$,
    $\Gamma(x : A) \vdash B \rhd ? : s_2$ *and* $\Gamma \vdash T \cong_\beta \Pi x^A.B$.
  - *If* $\Gamma \vdash s \rhd^+ N : T$, *then there is* t *such that* $N \equiv s$, $(s, t) \in Ax$, *and* $\Gamma \vdash T \cong_\beta t$ *or* $T \equiv t$.

This exchange of types will also be used in the proof of the *Church-Rosser* property to avoid building the right sets of sorts in *Rel* at some minor stage of the proof. However, we will use it extensively while proving that well-typed terms in PTSs can be correctly annotated into well-typed annotated terms in PTS_{atr}.

Lemma 3.2.10 (Type Correctness)**.** *If* $\Gamma \vdash M \rhd N : A$, *then there is* $s \in Sorts$ *such as either:* $A \equiv s$ *or* $\Gamma \vdash A \rhd ? : s$.

Theorem 3.2.11 (From PTS_{atr} to PTSs and PTS_e)**.**

1. *If* $\Gamma \vdash M \rhd N : A$ *then* $|\Gamma| \vdash |M| : |A|$,
   $|\Gamma| \vdash |N| : |A|$ *and* $|M| =_\beta |N|$.

2. *If* $\Gamma \vdash M \rhd N : A$ *then* $|\Gamma| \vdash_e |M| : |A|$,
$|\Gamma| \vdash_e |N| : |A|$ *and* $|\Gamma| \vdash_e |M| =_\beta |N| : |A|$.

Proof. This proof is much like the translation from PTS_e to PTSs: we have more typing in formations in PTS_{atr} than in PTSs or PTS_e, so we just need to remove the additional annotations. Since $\rhd$ has been designed to mimic the parallel reduction for β, it is quite easy to show that erased terms are still connected by typed or untyped β-conversion. □

Corollary 3.2.12 (Sort and Π-types incompatibility)**.** *It is impossible to prove that* $\Gamma \vdash \Pi x^A.B \cong_\beta s$ *for any* Γ, A, B, s*.*

Proof. The proof relies on a translation of the equality judgment $\Gamma \vdash \Pi x^A.B \cong_\beta s$ in the PTSs by erasure of the annotations with the first part of Theorem 3.2.11. The confluence of β-reduction forbids that $\Pi x^{|A|}.|B| =_\beta s$ in any way. □

At this point we need to recall what we said about the order we used to prove things in PTSs. We did not present any kind of confluence for PTS_{atr}. The reason is that, in a typed framework like PTS_e or PTS_{atr}, the *Confluence* and the *Church-Rosser* properties are a blocking step. Since they mix together typing and reduction, it is difficult to find a proof without involving the *Subject Reduction* of the system, and the proof of this theorem involves already knowing the Π-injectivity property (as required for PTSs in the previous section) which comes from *Confluence*: we need to break this loop.

3.2.3 The *Church-Rosser* Property in PTS_{atr}

The next step in the meta-theory is to prove the *Church-Rosser* property by proving that PTS_{atr} enjoys the *Diamond Property*:

Theorem 3.2.13 (Diamond Property)**.** *If* $\Gamma \vdash M \rhd N : A$ *and* $\Gamma \vdash M \rhd P : B$*, then there is* Q *such that*

$\Gamma \vdash N \rhd Q : A$	$\Gamma \vdash N \rhd Q : B$
$\Gamma \vdash P \rhd Q : A$	$\Gamma \vdash P \rhd Q : B$

We are trying to close the classic *Church-Rosser* diamond diagram in a typed way. In all previous attempts [Ada06, SH10], the main issue was to be able to close the cases involving an application constructor: APP/APP,

APP/BETA and BETA/APP. We lacked information about the co-domain (say D) of the application:

1. some types involved in the conclusion of those judgments are substituted (e.g. $D[N/x]$), so we lack the complete typing information for D.

2. some induction hypotheses over the co-domain of types do not always reflect the context of the hypothesis we actually have.

The first problem is "easily" solved by adding the D as an annotation. But it is this additional annotation that makes the second problem arise: it forbids us to use one of our induction hypothesis. During the proof, the induction hypothesis requires the context to be the same in both branches of the theorem and for the APP case, we needed to prove that it was actually the case.

In his proof, Adams relies on the *Uniqueness of Typing* which comes from the functionality, and in the semi-full case, we relied on the *Shape of Types* [SH10] to make the contexts match. To get rid of both constraints over the PTSs, we use here the new annotation in applications, that *forces* the context to match: now in the APP/APP case, co-domain contexts are syntactically the same, and in the other cases (where we actually perform a β-reduction), we have enough typing information to type the resulting substitution.

We will not give more details about the second issue here since we no longer face it (explanations and a concrete example can be found in [SH10]. However, since it is the first time that the new annotation comes in handy, we can now explain our choices for it.

The annotation is here to have a full remainder of the function space: if $\lambda x^A.M$ of type $\Pi x^A.B$ is applied to N of type A, we want to have both A and B available while looking at the β-redex. Our first attempt was to put *syntactically* the same A in the annotation, and thus allowing the reduction of the redex only if the annotation matches exactly the domain of the function. But this approach failed, and made us realized that we need to annotate by any A' convertible to A. However, this notion of conversion has to be more strict than our $\cong_\beta$ judgment: we need to enforce that each conversion step stays in the same sort, much like the equality judgments for PTS_e.

We could have used two different notions of conversion, one that cares about the type, and one that only cares about the types being sorts, but

the first one was only needed for this new annotation, and as soon as we proved *Confluence*, we will always break it into two multi-step reductions. Instead, we tried to find another "self-contained" notion of strict conversion, with the judgments we built so far around $\triangleright$, $\triangleright^+$ and $\cong_\beta$. Having a common expanded term satisfied all our requirements:

- all the steps between the domain and the annotation are well-typed, by the very same sort.
- we do not have to introduce a new kind of judgment.
- it behaves nicely in the proof of the *Church-Rosser* property.

The main reason why this choice behaves so nicely is that PTS_{atr} is a *reduction* system: it is directed. The domain and the annotation are always reduced in the same direction. Informally, if $A_0 \triangleright^+ A$ and $A_0 \triangleright^+ A'$, since both A and A' can only be reduced, we just have to append the new reductions steps to the sequences starting from A_0. In short, we never need to "guess" what is the common expanded term.

Doing so, the proof of the *Diamond Property* becomes quite straightforward by induction, since we pushed all the issues inside the new annotation. However, those issues did not disappear: we will have to face them when while proving that the annotations are correct.

3.2.4 Consequences of the *Church-Rosser* property

With the *Church-Rosser* property, we can finally settle with all the missing pieces of theory that we do not know how to prove directly in a typed framework:

Lemma 3.2.14 (Confluence)**.**
If $\Gamma \vdash A \cong_\beta B$*, there are* C, s, t *such that* $\Gamma \vdash A \triangleright^+ C : s$ *and* $\Gamma \vdash B \triangleright^+ C : t$*.*

Lemma 3.2.15 (Weak Π-injectivity for PTS_{atr})**.**
If $\Gamma \vdash \Pi x^A.B \cong_\beta \Pi x^C.D$ *then* $\Gamma \vdash A \cong_\beta C$ *and* $\Gamma(x : A) \vdash B \cong_\beta D$*.*

Since strong injectivity does not hold for PTS_{atr} (the same counterexample we used for PTS_e also works here), we stated a weaker form of injectivity. However, this Π-*injectivity* for $\cong_\beta$ along with the *Exchange of Types* properties are enough for the rest of the development.

Theorem 3.2.16 (Subject Reduction)**.**
If $\Gamma \vdash M \rhd ? : A$ *and* $M \to_p N$ *then* $\Gamma \vdash M \rhd^+ N : A$.

Proof. This is the first place where we encounter the difficulties that we postponed in the proof of *Church-Rosser* property. It is interesting to notice that we did not manage to prove the conclusion of *Subject Reduction* as a one step PTS_{atr} reduction: all the conversion we have to do to make the annotations in the application match forced us to have a multi-step version of the conclusion. However, this will not be a problem for the following proofs.

The proof is done by induction on $M \to_p N$: as usual, most cases are trivial. In the case of application congruence, some type conversions are required, but everything is directly available. However, if M is a β-redex which is reduced, we need to show that we have the right to do this reduction according to the typing rule BETA. We will make an extensive use of *Confluence* and *Exchange of Types* to show that everything is fine.

The situation is the following: we want to prove that

$$\Gamma \vdash (\lambda x^A.M_{\Pi x^{A'}.B'})N \rhd^+ M'[N'/x] : B'[N/x]$$

knowing that the β-redex is well-typed. By inversion, we have two choices: either the redex is typed as an application, or it is typed with the BETA rule. In the second case, we directly have all the information to conclude. However, in the first case, we need to use the *Generation* Lemma to retrieve typing information from the application and from the λ-term. We get the following judgments:

- $\Gamma \vdash A \rhd ? : s_1$, $\Gamma(x : A) \vdash M \rhd ? : B$ and $\Gamma(x : A) \vdash B \rhd ? : s_2$ where $(s_1, s_2, s_3) \in Rel$.
- $\Gamma \vdash A' \rhd ? : t_1$, $\Gamma(x : A') \vdash B' \rhd ? : t_2$ where $(t_1, t_2, t_3) \in Rel$.
- $\Gamma \vdash N \rhd ? : A'$ and $\Gamma \vdash \Pi x^A.B \cong_\beta \Pi x^{A'}.B'$.

Using Π-*injectivity*, we can show that $\Gamma \vdash A \cong_\beta A'$, but as we said before, this is not enough to trigger the reduction of the redex, since A and A' are not typed by the same sort, and we do not know any common expanded form for them. However, by *Confluence*, we can find common reduced terms for A and A' , and also for B and B':

- $\Gamma \vdash A \rhd^+ A_0 : s$ and $\Gamma \vdash A' \rhd^+ A_0 : t$.

- $\Gamma \vdash B \rhd^+ B_0 : s'$ and $\Gamma(x:A) \vdash B' \rhd^+ B_0 : t'$.

Using the *Exchange of Types*, we can replace s by s_1, t by t_1, s' by s_2 and t' by t_2. Doing so, we can prove that $\Gamma \vdash \lambda x^A.M_{\Pi x^{A'}.B'} N \rhd^+ \lambda x^A.M_{\Pi x^{A_0}.B_0} N : B'[N/x]$. With this new redex, we have everything at hand to fire the reduction and prove that
$\Gamma \vdash \lambda x^A.M_{\Pi x^{A_0}.B_0} N \rhd M[N/x] : B_0[N/x]$.

With (REDS-TRANS-ALT), and the *Substitution* Lemma, we can now glue both reductions and conclude the final case of *Subject Reduction*. □

3.3 Equivalence of PTS$_{atr}$ and PTSs

3.3.1 Confluence of the annotation process

The last step to prove the equivalence is to prove the correctness of annotations, i.e. to prove that every judgment $\Gamma \vdash M : T$ can be annotated into a valid PTS$_{atr}$ derivation $\Gamma^+ \vdash M^+ \rhd M^+ : T^+$ where $|\Gamma^+| \equiv \Gamma$, $|M^+| \equiv M$ and $|T^+| \equiv T$.

To do so, we need to show some basic properties of the annotation process. Since there are several ways to annotate a term, we face some difficult situations while performing induction. Let us go back to the simple example with the construction of Π-types with the PI rule:

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

By induction, we get that $\Gamma_1 \vdash A_1 \rhd A_1 : s_1$ and $\Gamma_2(x : A_2) \vdash B_2 \rhd B_2 : s_2$ with the equalities $|\Gamma_1| \equiv |\Gamma_2| \equiv \Gamma$, $|B_2| \equiv B$ and $|A_1| \equiv |A_2| \equiv A$. To build a Π-type from those two judgments, we need to relate Γ_1 to Γ_2 and A_1 to A_2 *in PTS$_{atr}$*. More precisely, we need to show that if two annotated types come from the same non-annotated term, and if they are well-typed in PTS$_{atr}$, they are equivalent in PTS$_{atr}$. With such a property, we would be able to state a similar lemma for contexts and prove that our annotation procedure is correct.

However, we have to recall that what we call here types are just terms typed by a sort, and their typing judgment may use β-redexes, which may involve "non-types". So we will state a more general lemma about the conversion of different annotated versions of a same PTS term.

Lemma 3.3.1 (Erased Confluence)**.** *If* $|M| \equiv |N|$, $\Gamma \vdash M \rhd ? : A$ *and* $\Gamma \vdash N \rhd ? : B$ *, then there is* R *such that*
$\Gamma \vdash M \rhd^+ R : A$ *and* $\Gamma \vdash N \rhd^+ R : B$*.*

Proof. The proof is done by induction on M, the only difficult part is again the application case:

$$M \equiv P_{\Pi x^{A_0}.D} Q,\ \ N \equiv P'_{\Pi x^{A'_0}.D'} Q' \qquad\qquad |P| \equiv |P'|,\ \ |Q| \equiv |Q'|$$

By generation, we get that P, P', Q and Q' are well-typed, so by induction, there are P_0, Q_0 such that:

$$\begin{array}{ll} \Gamma \vdash P \rhd^+ P_0 : \Pi x^C.D & \qquad \Gamma \vdash Q \rhd^+ Q_0 : C \\ \Gamma \vdash P' \rhd^+ P_0 : \Pi x^{C'}.D' & \qquad \Gamma \vdash Q' \rhd^+ Q_0 : C' \end{array}$$

and some additional information relating A_0 and A'_0 to C and C' depending on the way M was typed (BETA or APP).

In the functional case (where only one annotation is needed), this is quite trivial : thanks to the *Uniqueness of Types* applied to P_0 and Π-injectivity we get that $\Gamma(x : C) \vdash D \cong_\beta D'$. By *Confluence*, we get a common reduct D_0 for D and D', so the common reduct of M and N is $P_0\ {}_{D_0} Q_0$.

We need to be a little more subtle here: for the semi-full case (see [SH10, Siles & Herbelin, 2010]), we showed that terms can be classified in two families whose types have very particular shapes. Fortunately, the full generality of this classification is not needed here:

Lemma 3.3.2 (Weak shape of types)**.** *If* $\Gamma \vdash M \rhd ? : A$ *and* $\Gamma \vdash M \rhd ? : B$*, then:*

- *either* $\Gamma \vdash A \cong_\beta B$
- *or we are in one of the following cases:*
  1. *there are* U *and* V *such that* $\Gamma \vdash M \rhd \lambda x^U.V : A$ *and* $\Gamma \vdash M \rhd \lambda x^U.V : B$*.*
  2. *there is* s *such that* $\Gamma \vdash M \rhd s : A$ *and* $\Gamma \vdash M \rhd s : B$*.*
  3. *there is* U *and* V *such that* $\Gamma \vdash M \rhd \Pi x^U.V : A$ *and* $\Gamma \vdash M \rhd \Pi x^U.V : B$*.*

The proof of this lemma is quite trivial by induction, and relies on the fact that we have the annotation of co-domains at hand.

From now on, we will mainly focus on P_0: we can apply the previous lemma to it and, for the first part of the conclusion, conclude almost like the functional case. By generation, we also got a way to prove that $\Gamma \vdash A_0 \cong_\beta A'_0$, depending on the constructor used. By *Confluence*, we can get a common reduct A'', and use $P_0\ _{\Pi x^{A''}.D_0} Q_0$ to close the whole confluence lemma.

In order to apply the *Weak shape of types* to P_0, we need to show that P_0 appears on the left-hand side of an PTS$_{atr}$ typing judgment. However, at the moment, it only appears at the right of $\triangleright$. To be able to do this switch, we will rely on the *Right-Hand reflexivity* that we proved before. This step may seem trivial at this point, but it will be in important point when we will try to extend this proof to subtyping.

If we are in the second part of the conclusion, the only relevant case is the first one: since P_0 is typed by a Π-types, it can not reduce itself to a sort or another Π-type. The reason is because with the *Generation* lemma, we know that the type of a sort or a Π-type is always convertible to a sort. If they could be typed by a Π-type, we would end up having a judgment of the form $\Gamma \vdash \Pi x^A.B \cong_\beta s$ which is impossible due to Corollary 3.2.12.

In the last remaining case, there are U and V such that:

- $\Gamma \vdash P_0 \triangleright \lambda x^U.V : \Pi x^C.D$
- $\Gamma \vdash P_0 \triangleright \lambda x^U.V : \Pi x^{C'}.D'$

We just created a β-redex since P_0 is going to be applied, so this time, the common reduced term will be the result of the β-reduction initiated by P_0 instead of just a simple application.

Actually, we still need to show that we are allowed to reduce this redex, just as we needed to show it for *Subject Reduction*: this is the second place where we are facing quite technical points because of the new annotations. There are four different cases to handle here, depending on how M and M' are originally typed (by BETA or APP), but each can be closed by extensive use of *Confluence* and *Exchange of Types*. The main idea behind each case is the same, and follows this scheme:

$$
\begin{array}{llll}
\Gamma \vdash P_{\Pi x^U.D} Q & \rhd^+ & P_{0\ \Pi x^U.D} Q & : D[Q/x] \\
 & \rhd^+ & (\lambda x^U.V)_{\Pi x^U.D} Q & : D[Q/x] \\
 & \rhd^+ & V[Q/x] & : D[Q/x] \\
 & \rhd^+ & V[Q_0/x] & : D[Q/x] \\
\Gamma \vdash P'_{\Pi x^U.D'} Q' & \rhd^+ & P_{0\ \Pi x^U.D'} Q' & : D'[Q'/x] \\
 & \rhd^+ & (\lambda x^U.V)_{\Pi x^U.D'} Q' & : D'[Q'/x] \\
 & \rhd^+ & V[Q'/x] & : D'[Q'/x] \\
 & \rhd^+ & V[Q_0/x] & : D'[Q'/x]
\end{array}
$$

In the end, we manage to find a common reduct in each type without having to find a common reduct for the annotations, which concludes the proof of this lemma. □

3.3.2 Consequences of the Erased Confluence

With the general statement for all terms, we can now show what we needed about types and contexts:

Lemma 3.3.3 (Erased Conversion)**.** *1. If $|A| \equiv |B|$, $\Gamma \vdash A \rhd ? : s$ and $\Gamma \vdash B \rhd ? : t$ then $\Gamma \vdash A \cong_\beta B$.*

2. If $|\Gamma_1| \equiv |\Gamma_2|$ and $\Gamma_1 \vdash M \rhd N : A$, then $\Gamma_2 \vdash M \rhd N : A$.

Proof. The first statement directly follows from Lemma 3.3.1. The second is a consequence of the first one, by simple induction on the length of Γ_1. □

Now let us go back to the annotation of Π-types. With Lemma 3.3.3, we can derive the fact that $\Gamma_1 \vdash A_1 \cong_\beta A_2$ and $\Gamma_1 \equiv \Gamma_2$. By context conversion, we can exchange the contexts and we end up proving that $\Gamma_1(x : A_1) \vdash B_2 \rhd B_2 : s_2$, and so we can finally build the annotated judgment $\Gamma_1 \vdash \Pi x^{A_1}.B_2 \rhd \Pi x^{A_1}.B_2 : s_3$, with $|\Gamma_1| \equiv \Gamma$, $|A_1| \equiv A$ and $|B_2| \equiv B$.

By doing the same process for each constructor, we can now conclude the last missing piece of the whole equivalence process:

Theorem 3.3.4 (From PTSs to PTS_{atr})**.** *If $\Gamma \vdash M : T$, then there are Γ^+, M^+, T^+ such that $\Gamma^+ \vdash M^+ \rhd M^+ : T^+$, $|\Gamma^+| \equiv \Gamma$, $|M^+| \equiv M$ and $|T^+| \equiv T$.*

Proof. Since we have managed to prove *Subject Reduction* and Lemma 3.3.3, the proof is almost the same as for Adams' TPOSR. A few type exchanges are needed in the BETA case but it does not involve complicated nor technical things. □

Finally, all of this leads us to state that:

Theorem 3.3.5 (Equivalence of PTSs and PTS$_e$)**.**

1. $\Gamma \vdash M : T$ *iff* $\Gamma \vdash_e M : T$.

2. $\Gamma \vdash_e M =_\beta N : T$ *iff* $\Gamma \vdash M : T$*,* $\Gamma \vdash N : T$ *and* $M =_\beta N$.

Proof. This is just a combination of all the previous theorems:

- If $\Gamma \vdash_e M : T$, then by Theorem 3.1.2, we have $\Gamma \vdash M : T$.

- If $\Gamma \vdash M : T$, by Theorem 3.3.4 we know that

  $\Gamma^+ \vdash M^+ \rhd M^+ : T^+$ with $|\Gamma^+| \equiv \Gamma$, $|M^+| \equiv M$ and $|T^+| \equiv T$. By Theorem 3.2.11, $|\Gamma^+| \vdash_e |M^+| : |T^+|$ which is equal to $\Gamma \vdash_e M : T$.

- If $\Gamma \vdash_e M =_\beta N : T$, so we conclude by Theorem 3.1.2.

- If $\Gamma \vdash M : T$, $\Gamma \vdash N : T$ and $M =_\beta N$, by *Confluence*, there is P such that $M \twoheadrightarrow_\beta P$ and $N \twoheadrightarrow_\beta P$. By Theorem 3.3.4, there are Γ^+, M^+, T^+ such that $|\Gamma^+| \equiv \Gamma$, $|M^+| \equiv M$, $|T^+| \equiv T$ and $\Gamma^+ \vdash M^+ \rhd M^+ : T^+$. Let us consider P^+ such that $|P^+| \equiv P$ and $M^+ \twoheadrightarrow P^+$ (such a term always exists, the proof is a simple induction on the structure of M).

	$\Gamma^+ \vdash M^+ \rhd M^+ : T^+$	
$\Rightarrow$	$\Gamma^+ \vdash M^+ \rhd^+ P^+ : T^+$	(Subject Reduction)
$\Rightarrow$	$\Gamma \vdash_e M =_\beta P : T$	(Theorem 3.2.11 and TRANS)

  We do the same to conclude that $\Gamma \vdash_e N =_\beta P : T$, so by SYM and TRANS, we finally have $\Gamma \vdash_e M =_\beta N : T$.

□

3.3.3 Consequences of the equivalence

Now that we have a way to go from PTSs to PTS$_e$ (and the other way around), we can go back to the proof of *Subject Reduction* for PTS$_e$.

Theorem 3.3.6 (Subject Reduction for PTS$_e$)**.** *If* $\Gamma \vdash_e M : T$ *and* $M \rightarrow_\beta N$ *then* $\Gamma \vdash_e M =_\beta N : T$*.*

Proof. By using the first part of Theorem 3.3.5 and Theorem 3.3.4, there are Γ^+, M^+ and T^+ such that $\Gamma^+ \vdash M^+ \rhd M^+ : T^+$ and $|\Gamma^+| \equiv \Gamma$, $|M^+| \equiv M$ and $|T^+| \equiv T$. Let us consider N^+ such that $|N^+| \equiv N$ and $M^+ \rightarrow_p N^+$ (N^+ always exists, the proof is a simple induction on the structure of M). With such a term, and using Theorem 3.2.16, we can prove that $\Gamma^+ \vdash M^+ \rhd^+ N^+ : T^+$. By erasing the annotations using the last part of Theorem 3.2.11, we end up having $|\Gamma^+| \vdash_e |M^+| =_\beta |N^+| : |T^+|$ which is the exact result we wanted. □

The last missing piece of our development is to find the correct statement for injectivity of products in PTS$_e$. *Subject Reduction* for PTS$_{atr}$ relied on the *weak* Π*-injectivity* for $\cong_\beta$ and we choose such an equality to be able to state the *Generation* lemmas for PTS$_{atr}$. Since PTS$_{atr}$ is "enhanced" version of PTS$_e$ with additional annotations, that may be the correct presentation we were looking for:

Weak PTS$_e$ equality

$$\frac{\Gamma \vdash_e A =_\beta B : s}{\Gamma \vdash_e A =_\beta B} \qquad \frac{\Gamma \vdash_e B =_\beta A}{\Gamma \vdash_e A =_\beta B} \qquad \frac{\Gamma \vdash_e A =_\beta B \qquad \Gamma \vdash_e B =_\beta C}{\Gamma \vdash_e A =_\beta C}$$

This weaker form of equality enjoys some nice properties:

- If $\Gamma \vdash_e A =_\beta B$, then there are s and t such that $\Gamma \vdash_e A : s$ and $\Gamma \vdash_e B : t$.
- If $\Gamma \vdash_e A =_\beta B$, then $A =_\beta B$.
- This equality is compatible with conversion in PTS$_e$ context: if $\Gamma_1 \vdash_e A =_\beta B$ and $\Gamma_1(x : A)\Gamma_2 \vdash_e M : T$, then $\Gamma_1(x : B)\Gamma_2 \vdash_e M : T$.

All those properties are directly consequences of the usual equality for PTS$_e$.

With this equality, we can directly state some generation lemmas for PTS$_e$ without relying on the equivalence:

Lemma 3.3.7 (Generation Lemmas for PTS$_e$)**.** *Those properties are much like PTS$_{atr}$'s one, so we will only state the one that we will really need here:*

1. *If* $\Gamma \vdash_e \Pi x^A.B : T$ *then there are* s_1, s_2, s_3 *such that* $(s_1, s_2, s_3) \in Rel$, $\Gamma \vdash_e A : s_1$, $\Gamma(x : A) \vdash_e B : s_2$, *and* $T \equiv s_3$ *or* $\Gamma \vdash_e T =_\beta s_3$.

2. *If* $\Gamma \vdash_e \lambda x^A.M : T$ *then there are* s_1, s_2, s_3 *and* B *such that* $(s_1, s_2, s_3) \in Rel$, $\Gamma \vdash_e A : s_1$, $\Gamma(x : A) \vdash_e M : B$, $\Gamma(x : A) \vdash_e B : s_2$ *and* $\Gamma \vdash_e T =_\beta \Pi x^A.B$.

3. *If* $\Gamma \vdash_e M\ N : T$ *then there are* A *and* B *such that* $\Gamma \vdash_e M : \Pi x^A.B$, $\Gamma \vdash_e N : A$ *and* $\Gamma \vdash_e T =_\beta B[N/x]$.

Now that we have the *Generation Lemmas* and *Subject Reduction*, we can prove what we consider to be the *correct* statement for injectivity of products in PTS$_e$.

Corollary 3.3.8 (Weak Π-injectivity for PTS$_e$)**.** *If* $\Gamma \vdash_e \Pi x^A.B =_\beta \Pi x^C.D$ *then* $\Gamma \vdash_e A =_\beta C$ *and* $\Gamma(x : A) \vdash_e B =_\beta D$.

Proof. By using the properties of weak equality that we just stated, there are s_3 and s'_3 such that $\Gamma \vdash \Pi x^A.B : s_3$, $\Gamma \vdash \Pi x^C.D : s'_3$, and $\Pi x^A.B =_\beta \Pi x^C.D$. By Π-*injectivity* and *Confluence* for the usual untyped β, and *Generation* for PTS$_e$, we get:

- $A \twoheadrightarrow_\beta U \ {}_\beta\!\twoheadleftarrow C$ and $B \twoheadrightarrow_\beta V \ {}_\beta\!\twoheadleftarrow D$
- $\Gamma \vdash A : s_1$, $\Gamma \vdash C : s'_1$, $\Gamma(x : A) \vdash B : s_2$ and $\Gamma(x : C) \vdash D : s'_2$ for s_1, s'_1, s_2, s'_2 such that $(s_1, s_2, s_3) \in Rel$ and $(s'_1, s'_2, s'_3) \in Rel$.

By using *Subject Reduction for PTS$_e$*, we get that $\Gamma \vdash_e A =_\beta U : s_1$, $\Gamma \vdash_e C =_\beta U : s'_1$, $\Gamma(x : A) \vdash_e B =_\beta V : s_2$ and $\Gamma(x : C) \vdash_e D =_\beta V : s'_2$. It is now easy to glue everything together to obtain $\Gamma \vdash_e A =_\beta C$ and $\Gamma(x : A) \vdash_e B =_\beta D$. □

The "good" notion of equality

The two following points are the main reasons why we think that this notion of "weak" equality is the *correct* one for types:

- This proof of injectivity holds for *any* PTS$_e$, even the non-functional ones or the ones that do not enjoy normalization.

- The *Weak* Π-*injectivity* for PTS_e is enough to prove *Subject Reduction* in the usual way (by a direct induction).

Until now, the equivalence between *Subject Reduction* and the *Weak* Π-*injectivity* for PTS_e was known, but we had no direct proof of any of them. With the PTS_{atr} type system, we successfully proved *Subject Reduction* and thus managed to prove the injectivity as its consequence. As far as we know, all the attempts to build a direct proof of the injectivity of products have failed. The PTS_{atr} system is significantly more complicated then the usual presentation of PTS_e but it contains much more useful information. Further investigation should concentrate on PTS_{atr} since most of its properties directly apply to PTSs and PTS_e.

Chapter 4

Formalization in *Coq*

Contents

4.1 Formal proof: paper or computer ? 92
4.1.1 What is a formal proof ? 92
4.1.2 Automatic resolution and induction schemes . . . 93
4.2 Encoding PTSs in a proof assistant 95
4.2.1 Questions about encodings of binders 95
4.2.2 Higher order encodings 97
4.2.3 Our final choice: de Bruijn indices 99

Now that we proved the equivalence for any kind of PTSs, we can concentrate our efforts to achieve the same result with more complex systems than PTSs. The next chapter will be about such possible extensions of the proof. By considering type systems that provide more expressiveness than PTSs, we have to change the core definitions of our development (especially the reduction rules and the typing judgments), and thus prevent ourselves to directly use our previous results. Such modifications will force us to redo the whole meta-theory for any new system.

However, as we previously said, we formalized the whole proof of equivalence within the proof assistant *Coq*. Since we are currently interested in *extensions* of PTSs, we could reuse our formalization as a basis from where to start, and try to check what parts of the development easily scale and what parts need a serious upgrade. In this chapter, we are going to detail

some parts of the formalization techniques we used to check all these results within the proof assistant *Coq*[1], as a full-scale example of some of the currently available techniques in computer assisted reasoning.

4.1 Formal proof: paper or computer ?

4.1.1 What is a formal proof ?

One of the biggest achievement of the late 19th century is the work of Frege about mathematical proofs. He introduced a formalism expressive enough to (gradually) convince most mathematicians that his language of proofs could capture the vast majority of mathematics. A formal proof is a sequence of symbols linked together with inference rules. It is usually more convenient to represent them as a tree-shape data structure, where the nodes of the tree are the inference rules and the leaves are the axioms of the system. The validity of such proofs is done by checking that each symbol and each rule is correctly used with respect to a particular logical system (the "meaning" we attach to the proof). In the late 1960's, de Bruijn developed the first tool to automatically check the correctness of inference rules, called *Automath*, which was quickly followed by the *Nqthm* theorem prover by Boyer and Moore. They exemplified that proofs can be efficiently checked, stored, and rechecked at will by a computer. Another advantage is that, by mechanically checking proofs, most usual mistakes disappear, like forgetting a subcase of a proof, applying a hypothesis to an incorrect argument or making a mistake in a numeric computation.

This formalism is considered more formal and more detailed on the contrary of the usual "pen and pencil" approach, which leaves statements like "this is trivial" or "all the other cases are done in the same way" to the intuition of the reader, without entering in the exact details that would make a proof absolutely unquestionable. Nonetheless, being able to interpret such sentences is still a crucial issue for the future of proof assistants.

Writing a formal proof is also a good way to be sure that nothing has been forgotten. Even if sometimes, stating that "a simple induction is enough to prove the lemma", such behavior can lead to small mistakes, that could have been avoided by the rigor of formal proof. The *Type Correctness* (see

[1] In this chapter, *Coq* has to be understood as *Coq* version 8.3.

Lemma 2.1.7) is one of the best examples of a "not so wrong" lemma that we found in the literature. As we said at the time, we are tempted to state the following lemma:

$$\text{If } \Gamma \vdash M : T\text{, then there is } s \text{ such that } \Gamma \vdash T : s \quad (1)$$

This particular statement can be found in a few papers about PTSs and, disregarding the quality of these works, it is notably wrong in the general case since some PTSs have *top sorts* (like *Type* in the *Calculus of Constructions*) which are not typed by any sort. The proof is "a simple induction" on the structure of the judgment $\Gamma \vdash M : T$, but the rigor induced by the use of a formal proof would have spotted that when M is a sort, (1) is not always true.

4.1.2 Automatic resolution and induction schemes

The kind of rigor which is required by formal proofs may seem reluctant at first, by the amount of "administrative" things we need to provide, things that were usually left aside to the intuition of the reader. However, since the development of the first proof assistants, some improvement have been made towards *automation*. Besides checking the validity of a proof, a proof assistant can also provide other tools to ease the formalization process and help the user to focus on the really meaningful parts of his work. As an example of what is possible, *Coq* provides several ways to use automation within the scripts, like heuristics for automatic resolution of goals, automatic schemes generation for inductive types, mathematical computation, type classes...

In this development we only used the first two possibilities. Automatic resolution is mainly used to solve "easy" parts of the development. For example, in the *Type Correctness* lemma, the only relevant cases are the typing of variables and sorts. For all the other cases, we would have written something close to "trivial by applying the induction hypothesis". This is the kind of automation that *Coq* can provide, by the way of resolution tactics like `auto, eauto, firstorder, intuition` or `tauto`. Intuitively, these tactics use the available hypotheses of the current goal along with some previous knowledge that the user stored in *hint databases* to do a proof search in the same way *Prolog* would have done.

This use of automation is an appealing thing that allows the user to focus only on the difficult parts of its development. The resulting scripts are clearer

since we can directly see what was the interesting part of a proof, but that also means that the other parts are "hidden" to the user. In our opinion, this behavior should be saved for a time when a proof is really well understood, not while it is still under development. The main reason is that, during the design of a system, we often need to go back to the very first definitions and slightly modify them. After such modifications, we can simply run again our scripts and see what fails and what succeeds. However, if a failure happens during an automatic resolution (e.g. `auto` previously solved a goal and can not do it anymore), we will not have any meaningful information to patch the proof. On the contrary, when a proof is finally done and correctly understood, some use of automation is clearly a way to enhance the final version of this proof to put forward its core and hide the less relevant parts.

Some tactics can be used to give a more robust structure to a proof script, by failing if the current goal is not completely solved. Tactics like `assumption` or `solve` behave pretty much like *check-points*, that fail if the current goal is not completely solved by the tactics. This way, we can explicitly mark the end of a goal in our scripts, and any modifications that may break the proof should fail at more meaningful places.

The second point is more about the confidence we have in ourselves. Let us consider the following inductive type named `Term` that we use to encode the terms of syntactic PTSs:

```
Inductive Term :  Set:=
   | Var :  Vars -> Term
   | Sort :  Sorts -> Term
   | App :  Term -> Term -> Term
   | Pi :  Term -> Term -> Term
   | La :  Term -> Term -> Term
.
```

To perform a proof by induction on an object of type `Term`, we need an induction principle for this type. *Coq* is able to automatically derive it for us each time we define a new inductive type, but in the case of `Term`, it is still easy to do "by hand" since `Term` is a quite simple tree-shaped datatype. What happens if we face more complicated structures, like the mutual family `wf/typ` (for PTSs) or the even more complicated one made of `wf/typ/typ_reds` (for PTS_{atr})? Building a correct mutual inductive princi-

ple for these kinds of type families is a difficult exercise prone to introduce mistakes. *Coq* also provides a way to derive such a mutual principle, but we need to specifically ask it with the `Scheme Induction` and `Combined Scheme` command. This way, we can ask *Coq* to build these complex schemes and check that they are effectively correct, which is much more time saving that trying to build the mutual statements ourselves. However, in particular situations, we still need to provide the induction principle by hand because the automatic schemes provided by *Coq* does not exactly suit our needs (see the *typ_annot.v* file for a particular example).

4.2 Encoding PTSs in a proof assistant

4.2.1 Questions about encodings of binders

In order to have more confidence in our proofs, we then chose to formalize our work within a proof assistant. Even before choosing a particular proof assistant, we need to choose a way to encode all the structures that appear in our work. As we did in all the previous chapters, we first need to give the definitions that describe the core of our development, which can be restricted to four main categories: sorts, terms, contexts and typing judgments.

The most delicate part of these definitions is the way we deal with *binders*. Let us consider the following term:

$$\begin{aligned} & f\ (\lambda x.g\ x)\ x \\ =_\alpha\ & f\ (\lambda z.g\ z)\ x \end{aligned}$$

If we want to substitute the *free* occurrence of the variable x by a term M, the result will be $f\ (\lambda x.g\ x)\ M$, leaving the bound occurrence of x unchanged since we deal with *capture avoiding* substitution. Depending on the way we are going to choose to encode binders, proving that the substitution is correct and reasoning modulo α-conversion as we did in the previous sections can become rather complicated. These two problems are known to be difficult and have already been studied a lot [Sch24, dB72, Bar84]. The *POPLmark challenge* [POP] is a good example of the practical use of several solutions for handling binders, used to solve a common formalization problem.

The most basic solution is to use proper names as variables, like strings. This way, it is easy to write terms "by hand", but reasoning modulo α-conversion will require extra work since we need to explain to the proof

assistant why $\lambda x.x$ and $\lambda y.y$ are the same term (modulo α-conversion). The substitution has to deal with a freshness condition to avoid the capture of variables, which will also complicate the work of formalization.

Another solution, close to the first one, tries to avoid these problems by enforcing a syntactical separation between bound and free variables and using two separate sets of names. This solution is known as the *locally named* representation first implemented by McKinna and Pollack [MP93, MP99]. In the case of the substitution of a bounded variable, capture could still happen but in practice, such substitutions were always used in a safe way, avoiding capture.

A canonical solution of both issues is the *de Bruijn* notation [dB72], which replaces all names by indices that directly link a variable to its binding λ-abstraction (see Fig. 4.1 for a simple illustration). Free variables are the ones

(λxλy.x)(λx.x)   λxλy. y (λx.x) x
(λ λ .1)(λ .0)   λ λ . 0 (λ .0) 1

Figure 4.1: de Bruijn indices examples

whose indices are greater than the longest sequence of λ-abstraction, which means that they are not bounded by any abstraction in the term but that they are "bounded by the enclosing context". This way, there is no problem to deal with α-renaming since there is a unique representation of closed terms, hence no ambiguity anymore. Furthermore, there is a simple and correct way to specify a capture avoiding substitution, by using a *shift* [ACCL91] (also known as *lift* [Hue02]) operator which does not require to check any freshness condition[2].

Such encoding of terms is completely uniform, and has proved its efficiency by having been used to successfully formalize several meta-theories (see [BW97] for example). The attempts to the POPLmark challenge that went the further in the formalization of the problem are actually using de

[2]See the implementation of `subst_rec` in the file *ut_term.v*.

Bruijn indices (see the solutions of Berghofer and Vouillon in [POP]). This solution is quite elegant to reason about binders, but writing a term "by hand" is a difficult exercise, mainly because finding the correct index can become complicated in big terms. This notation is also difficult to read since we removed the names of the variables on the abstractions.

The *locally nameless* approach tries to have the good part of both names for free variables and de Bruijn indices for bounded variables [ACP+08]. This way, substitution can be described intuitively without having to rely on a *shift* operator, α-equivalence is made trivial thanks to de Bruijn indices, and manually handling terms is less complicated than with only indices. However, to reason about terms in locally nameless form, we still have to deal with freshness side-conditions at some point of the development, to avoid collisions of names in the typing judgments. In addition to that, every time we want to "look" under a λ-abstraction, we need to *open* the body of the abstraction by replacing the first de Bruijn index by a fresh name, and thus we need to parse the whole term, which seems as costly as using the *shift* operator: we loose the uniformity of the de Bruijn indices representation and we still have to deal with troublesome operators to correctly handle bound variables and the freshness condition.

All these encodings are aiming toward the same goal: allowing the user of a proof assistant to write and read proofs without having to provide more than he would have to while writing a proof with "pencil and paper". *Nominal Isabelle* [Urb08, HU10] is an implementation built on top of *Isabelle/HOL* [NPW] that aims at this particular purpose. It provides the use of names for bindings, and provides a way to safely handle α-conversion, capture-avoiding substitution and a proper induction principle, so that the user does not have to deal with this part of the encoding. This solution is, according to us, maybe the most elegant solution to deal with binders without having to deal with the issues we mentioned previously.

4.2.2 Higher order encodings

In all the previous solutions, terms and binders were all described as first-order datatypes. Proof assistants can also be used as programming languages, and embed *functions*. Since the λ-abstraction is a description of such functions, we could be tempted to encode these abstractions using the functions

of our language. Here is an example of how we could have done it in *Coq*[3] for the *untyped* λ*-Calculus*:

first-order encoding	higher-order encoding
`Inductive fTerm :=`	`Inductive hTerm :=`
`| fVar:  Vars -> fTerm`	
`| fAbs:  Vars -> fTerm -> fTerm`	`| hAbs:  (hTerm -> hTerm) -> hTerm`
`| fApp:  fTerm -> fTerm -> fTerm`	`| hApp:  hTerm -> hTerm -> hTerm`
`.`	`.`

```
Definition fId := fAbs "x" (fVar "x").
Definition hId := hAbs (fun x => x).
```

As you can see, `fId` is just a first-order datatype that can be parsed as the sequence of symbols `fAbs, "x", (, )` and `fVar`, whereas `hId` uses a *Coq* level function to build the abstraction. This kind of encoding is known as *Higher Order Abstract Syntax*, or HOAS for short. We got rid of this kind of solution quite early in our choosing process since HOAS escapes the framework of finite proofs. Even if `hId` can be parsed as the sequence made of `hAbs, fun, x, =>, (`, and `)`, by allowing functions inside our encoding, we capture *any* kind of functions that exists in the logic (e.g. in *Coq*, that would allow the use of any *Coq* function), without limiting to the ones that we can build with just our encoding of the λ-terms. Such approach would not be a correct encoding of the λ-calculus since it would capture *more* than just the λ-terms.

However, some languages like *Twelf* [PS] have restrictions on the domain of their functions which allow such an approach. The fragment of computable functions is weaker than in other languages and exactly represent the functions of the λ-calculus. Recent work in HOAS made it accessible to proof assistants that do not embed it naturally, like *Coq* or *Isabelle/HOL* thanks to *Hybrid* [FMCM].

[3] The type `hTerm` is actually rejected by *Coq* as a not well-formed inductive type, it is described here just for illustrating our point.

4.2.3 Our final choice: de Bruijn indices

Coq in Coq [BW97] and previous formalization of PTSs [Bara] done by Bruno Barras in *Coq* seemed a good starting point for our own work. Barras also formalized the work of Adams (the particular case of TPOSR for the *Calculus of Constructions* can be found at [Barb]) which is at the heart of this dissertation, so we chose to follow his path and use the proof assistant *Coq* with an encoding based on *de Bruijn* indices. As we said in our survey about the encoding techniques, it is quite difficult to write particular λ-terms with de Bruijn indices, but previous formalizations have shown that they are an efficient way to deal with families of terms (all of our propositions are universally quantified over terms or judgments, like in "$\forall \Gamma, M, T$ if $\Gamma \vdash M : T$ then ...").

In fact, from our experience during this past three years, we almost never had to directly deal with de Bruijn indices, and by following Barras encoding of contexts, we managed to deal with these indices as if they were an abstract type or simple strings, everywhere but in two lemmas: *Weakening* and *Substitution*. De Bruijn indices can describe bound variables without any ambiguity, but free variables have to be dealt with carefully. In fact, there is no free variable in our work, every time a variable is not bound by a λ-abstraction, it is bound in the context of the derivation (see Fig. 4.2 for an illustration of de Bruijn indices with contexts). In the following, by saying that a variable is "free", we mean that it is not bound by a λ-abstraction, but simply by the context of the judgment.

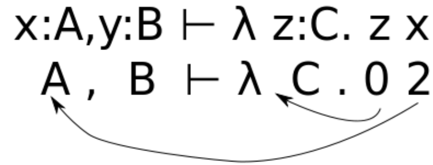

Figure 4.2: Context and de Bruijn indices

If we could enforce such an invariant in our formalization, we could completely forget to check that all of the judgments we consider are closed. It is quite easy to see that the only place where such check has to be done is in the typing rule for variables:

$$\frac{\Gamma_{wf} \qquad \Gamma(x) = A}{\Gamma \vdash x : A} \text{VAR}$$

The second premise should read as "x is in the domain of Γ and its type is A", and behaves like a *lookup* function. Barras developed such a lookup function[4] for de Bruijn indices which has the same behavior, allowing us to state our typing rule for variables *in the same way* as we would have done on paper, hiding all the de Bruijn indices:

```
cVar :  forall Γ A x, wf Γ -> A ↓ x ⊂ Γ -> Γ ⊢ x : A
```

With this function, we are sure that every time we use the `cVar` rule, the context is large enough to correctly bind the variable x. Now that we have hidden the use of de Bruijn indices in the typing judgments, the only places left where we may need to directly deal with indices manipulation are inside the lemmas of our development. As we said, things are in fact easier than we first expected: we only have to directly deal with de Bruijn indices in two lemmas. The first one is the *Weakening* lemma, where we want to add an additional hypothesis in a context. By doing so, we have to modify some "free" indices in order to take into account the new shape of the context. Thanks to the *shift/lift* operator[5], it is quite easy to state this lemma, and once the proof is done, it can be used without thinking of the manipulations that happen inside.

Here is the actual statement of the *Weakening* lemma. The notation `ins_in_env Δ A x Γ Γ'` is an inductive type that describes the insertion of a type in a context and takes care of the de Bruijn manipulations: assuming A is a well-formed type in Δ and that Γ is of the shape $\Delta\Delta_0$, then $\Gamma' \equiv \Delta(x : A)\Delta_O$.

```
Notation "!  s" := (Sort s).
Notation " M ↑ n # k " := (lift_rec n k M).

Theorem weakening:  forall Γ M T, Γ ⊢ M : T ->
   forall Δ A s x Γ', ins_in_env Δ A n Γ Γ' -> Δ ⊢ A : !s ->
   Γ' ⊢ M ↑ 1 # n :  T ↑ 1 # n.
```

[4]In the *Coq* development, we write `A ↓ x ⊂ Γ.` for $\Gamma(x) = A$.

[5]In the *Coq* development, this operator is called `lift_rec`.

In the particular case where the insertion happens at the beginning of the context (which is the case most of the time), the statement is even easier to read:

```
Theorem thinning:  forall Γ M T, Γ ⊢ M : T ->
   forall A s, Γ ⊢ A : !s -> A::Γ ⊢ M ↑ 1 :  T ↑ 1.
```

The second one is the *Substitution* lemma, where we want to replace a "free" variable by a term of the same type. This action also needs to tweak some indices (the substitution actually replaces one index by a term, so all the indices above need to be adjusted) but again, once the proof is over, we do not have to remember what happened inside. Moreover, all traces of indices have disappeared from the statement itself, which exactly looks like a statement in a nominal encoding. All the manipulations of de Bruijn indices have been wrapped into the functions that perform the substitions.

Here is the actual statement for the *Substitution* lemma within *Coq*. The notation `sub_in_env Δ P A x Γ Γ'` is an inductive type which means that, assuming $\Delta \vdash P : A$ is a valid judgment and Γ is of the shape $\Delta(x : A)\Delta_0$, then $\Gamma' \equiv \Delta(\Delta_0[P/x])$.

```
Theorem substitution :  forall Γ M T Δ P A Γ' x,
      Γ ⊢ M : T -> Δ ⊢ P : A -> sub_in_env Δ P A x Γ Γ' ->
      Γ' ⊢ M [ x ← P ] : T [ x ← P ].
```

Except from these two places, de Bruijn variables were transparent, and behaved as if they were an abstract type. Our experience shows that, as long as we only deal with general statements that quantify over all terms of some kind, we never need to actually build a particular term by hand, and thus we avoid the major troubles of the use of de Bruijn indices while retaining only the interesting points: uniformity of the encoding, easy capture avoiding substitution, and α-conversion for free.

Part IV

Conclusion and Further Research

Chapter 5

Extensions of PTS

Contents

5.1 Sorts, order and subtyping **107**

5.2 Toward a typed conversion for CC_ω **110**

5.2.1 The straightforward approach 110

5.2.2 Other attempts and possible leads 112

5.3 Other leads for future investigations **114**

In the previous chapters, we have focused our efforts towards Pure Type Systems for two main reasons:

1. We wanted to have the most general result we could achieve in order to save us some work to adapt the results to different implementations: since PTSs are a framework that describes the core of many typing systems, we thought it was a good call.

2. PTSs have been used as a starting point for the elaboration of the theory behind several proof assistants, which is our ultimate goal.

Right now, we have only considered β-equality as our conversion rule. In order to get closer to the actual systems implemented in real software, we can explore several different ways: we can extend the typing system itself, extend the syntax of terms, or extend the power of the conversion. Since we are not interested in using this language in practice (at least not yet), extending its syntax (with pairs or primitive integers for example) seems pointless at the

moment.

Extending the power of conversion was our first motivation to investigate this question: we wanted to define a type system that can deal with $\beta\eta$-conversion. Dealing with η-conversion is a difficult task, and all the results we are aware of rely on normalization properties (see [Geu93, Wer94]) The main issue of adding η to the conversion is that *Church-Rosser* is no longer true on raw terms[1], it is only true on well-typed terms. This property can be clearly illustrated by the following counterexample due to Nederpelt:

$$\begin{array}{lcl} \lambda x^A.(\lambda y^B.y)x & \to_\beta & \lambda x^A.x \equiv id_A \\ \lambda x^A.(\lambda y^B.y)x & \to_\eta & \lambda y^B.y \equiv id_B \end{array}$$

If A and B are not equal, id_A and id_B are two separate normal forms, and so the Church-Rosser diagram can not be closed.

As we said in the introduction, η-conversion was our first motivation to investigate the use of typed conversion. If we consider η as an expansion, we need to check that the term is actually a function (we do not want to expand every term), and we also need to find the correct type to put in the domain: we need to have the type of the term when performing the expansion. We did a few tries in this direction, but to this day, all of our attempts at proving Church-Rosser for such a typed β-reduction/η-expansion system have failed. The main issue we faced during our attempts with η-expansion was to find a correct statement for the *Generation* lemmas. Now that we are allowed to expand terms which are typed by a Π-type, variables and applications can be transformed to λ-abstraction by the reduction. To this day, we did not find any formulation of these *Generation* lemmas that where suitable enough to be used during the proof of Church-Rosser.

The final direction we can choose is to extend the level of types, by adding inductive types or cumulativity. The first option does not seem too complicated since it looks almost orthogonal to all the blocking steps we encountered (even if we did not yet investigate the behavior of the guard condition for the good formation of recursion on the syntax of annotated terms). Instead, we have tried to extend Pure Type Systems with cumulativity in order to focus on systems that can not be directly encoded into regular PTSs.

[1]A raw term is a term which type in unknown, so it can be typable, or not.

5.1 Sorts, order and subtyping

As a first step towards the full meta-theory of *Coq*, we first turned to the *Calculus of Constructions with Universes* [Miq01] (or CC_ω) whose meta-theory is quite close to a full PTS with a simple universe hierarchy[2].

Until now, we only considered systems where the conversion was an *equality*. This presentation is enough most of the time, but one could want to give an *order* to the set of sorts, to build stratified layers of types. The type theory of Russell and Whitehead [WR27] first introduced such a hierarchy in the set of types to avoid the paradox (also known as the "set of all sets" paradox) that Russell found in Frege's set theory. Another example of such layers is the type theory of Martin-Löf, built from the "$Type : Type$" system [ML71] which had been shown inconsistent by Girard [Gir71]. To fix this problem, Martin-Löf presented a version with a hierarchy of universes [ML84], which has since then inspired a lot of other type systems.

Actually, most of CC_ω can be directly embedded in a PTS by a wise choice of $\mathcal{A}x$ and $\mathcal{R}el$, but the cumulativity of Π-types prevent us to make a complete embedding of CC_ω inside the PTSs. In order to add subtyping to PTSs, we need to modify the conversion rule to allow the subtyping of sorts and its natural extension to products:

Sorts and order in CC_ω

$$\begin{array}{lcl} Sorts_{CC_\omega} & ::= & \{Prop\} \cup \{Type_i \mid i \in \mathbb{N}\} \\ \mathcal{A}x_{CC_\omega} & ::= & \{(Prop, Type_0)\} \cup \{(Type_i, Type_{i+1}) \mid i \in \mathbb{N}\} \\ \mathcal{R}el_{CC_\omega} & ::= & \{(s, Prop, Prop), (Prop, s, s) \mid s \in Sorts_{CC_\omega}\} \cup \\ & & \quad \{(Type_i, Type_i, Type_i) \mid i \in \mathbb{N}\} \end{array}$$

The definition of $\mathcal{A}x_\omega$ allows use to consider a total order on sorts, defined as:

- $Prop < Type_0 \qquad \forall i \in \mathbb{N}, Type_i < Type_{i+1}$
- $\forall s, t, u \in Sorts_{CC_\omega}$, if $s < t$ and $t < u$, then $s < u$.

[2]The definition of CC_ω in [Miq01] is quite ambiguous about the definition of subtyping and cumulativity. In the following, CC_ω has to be understood as the pure part of the *Extended Calculus of Constructions* [Luo89] (without Σ-types). This is the definition that is commonly found in the literature.

$$\frac{}{Prop \leq_\beta Type_0} \qquad \frac{i \in \mathbb{N}}{Type_i \leq_\beta Type_{i+1}}$$

$$\frac{A =_\beta C \qquad B \leq_\beta D}{\Pi x^A.B \leq_\beta \Pi x^C.D}$$

$$\frac{A =_\beta B}{A \leq_\beta B} \qquad \frac{A \leq_\beta B \qquad B \leq_\beta C}{A \leq_\beta C}$$

Figure 5.1: Subtyping relation of ECC

Subtyping in CC_ω is described in Fig. 5.1. The central rule is the one that prevent to encode it as a PTS: two Π-types can be related by the subtyping relation *without* being related by the $=_\beta$ equality. As for PTSs, we can define the computation on raw terms, without any typing information, and therefore keep a clean distinction between the behavior of cumulativity and typing judgments: cumulativity *does not* rely on any typing rule. The typing rules of CC_ω can be found in Fig. 5.2.

As you can see, the only difference with PTSs besides instantiation of $\mathcal{A}x$ and $\mathcal{R}el$ is the CONV rule which now involves cumulativity on top of equality. This system has been designed (together with Σ-types and there called *Extended Calculus of Constructions*, ECC for short) by Luo in is PhD thesis [Luo89], where he proved central properties like *Confluence*, *Subject Reduction* or *Strong Normalization*.

One interesting thing pointed out by Pollack when looking for a type checking algorithm for semi-full PTSs and for ECC, is that ECC can be changed into an equivalent *syntax-directed system*: it is possible to embed the CONV rule inside the other rules, so that it only remains exactly *one* typing rule for each type constructor. The exact same process can be applied to CC_ω, which makes the construction of a correct type for a particular term intuitive, one only has to apply the corresponding rule. Some examples of syntax-directed systems can be found in [vBJMP93], together with explanations on how to construct them from usual systems.

Even if the subtyping relation strictly extends the expressiveness of the

$$\frac{}{\emptyset_{wf}}\ \text{NIL} \qquad \frac{\Gamma \vdash A : s \qquad x \notin Dom(\Gamma)}{\Gamma(x : A)_{wf}}\ \text{CONS}$$

$$\frac{\Gamma_{wf} \qquad (s,t) \in \mathcal{A}x_{CC_\omega}}{\Gamma \vdash s : t}\ \text{SORT} \qquad \frac{\Gamma_{wf} \qquad \Gamma(x) = A}{\Gamma \vdash x : A}\ \text{VAR}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : s \\ \Gamma(x : A) \vdash B : t \qquad \Gamma(x : A) \vdash M : B\end{array}}{\Gamma \vdash \lambda x^A.M : \Pi x^A.B}\ \text{LAM} \qquad \frac{\begin{array}{c}\Gamma \vdash A : s \\ \Gamma(x : A) \vdash B : t \qquad (s,t,u) \in \mathcal{R}el_{CC_\omega}\end{array}}{\Gamma \vdash \Pi x^A.B : u}\ \text{PI}$$

$$\frac{\Gamma \vdash M : \Pi x^A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}\ \text{APP} \qquad \frac{\begin{array}{c}A \leq_\beta B \\ \Gamma \vdash M : A \qquad \Gamma \vdash B : s\end{array}}{\Gamma \vdash M : B}\ \text{CONV}$$

Figure 5.2: Typing Rules for CC_ω

typing system, CC_ω still enjoys almost the same property about *Shape of Types* than PTSs: the only difference is that, in the Ts case, the two trailing sorts can be *strictly ordered*:

Theorem 5.1.1 (The Shape of Types in CC_ω).
If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$ then:

- *either $M \in Tv$ and $A =_\beta B$*
- *or $M \in Ts$ and there are Δ, s and t such that $A \twoheadrightarrow_\beta \Pi\Delta.s$, $B \twoheadrightarrow_\beta \Pi\Delta.t$.*
- *Moreover, in the previous case, either $s < t$ or $t < s$.*

At this point, we are pretty confident that the equivalence between PTSs and PTS$_e$ can be extended to a system with subtyping like CC_ω, but this is not as easy as it sounds. In the next section, we are going to detail some of our most promising attempts toward this result.

5.2 Toward a typed conversion for CC_ω

5.2.1 The straightforward approach

The syntax of CC_ω is close enough to the syntax of PTSs to be able to build a first system, $CC_{\omega\,atr}$, in the same way we built PTS_{atr} (see Fig. 3.3) from PTSs: starting form PTS_{atr}, we remove the rules RED and EXP, and replace them by the following "typed" subtyping relation:

$$\frac{\Gamma \vdash M \rhd N : A \qquad \Gamma \vdash A \leq_\beta B}{\Gamma \vdash M \rhd N : B}\ \textsc{sub}$$

$$\frac{\Gamma_{wf}}{\Gamma \vdash Prop \leq_\beta Type_0} \qquad \frac{\Gamma_{wf} \qquad i \in \mathbb{N}}{\Gamma \vdash Type_i \leq_\beta Type_{i+1}}$$

$$\frac{\Gamma \vdash A \cong_\beta C \qquad \Gamma(x : A) \vdash B \leq_\beta D}{\Gamma \vdash \Pi x^A.B \leq_\beta \Pi x^C.D}$$

$$\frac{\Gamma \vdash A \cong_\beta B}{\Gamma \vdash A \leq_\beta B} \qquad \frac{\Gamma \vdash A \leq_\beta B \qquad \Gamma \vdash B \leq_\beta C}{\Gamma \vdash A \leq_\beta C}$$

Figure 5.3: Subtyping relation of $CC_{\omega\,atr}$

This system is close enough from PTS_{atr} to be able to prove that it enjoys all the same properties: we were able to successfully prove that $CC_{\omega\,atr}$ enjoys the *Church-Rosser* property, the *injectivity of* Π-*types* and *Subject Reduction*. The only difficulty here is to be cautious during the proofs because of all the dependencies: $\rhd$-judgments depend on $\rhd^+$-judgments and $\leq_\beta$-judgments, which depend on $=_\beta$-judgments, which depend on $\rhd$-judgments. This is the kind of place where using *Coq* as a safeguard for well-formed induction was handy. The proofs are the same as for the usual PTSs, even if CC_ω is actually *full*: the cumulativity added some constraints during the proof of Church-Rosser that strictly requires to have the full annotation on applications, not just the co-domain like it was the case for full PTSs.

However, until now, every attempt has failed at proving that CC_ω can be annotated in $CC_{\omega\,atr}$ using the same process as for PTSs:

Lemma 5.2.1. *Erased Conversion does not hold in $CC_{\omega\,atr}$.*

Proof. Let us build a counterexample to the *Erased Confluence* (Lemma 3.3.1) for $CC_{\omega\,atr}$:
$\Gamma = (f : Prop \to Prop, x : Prop)$ $\quad A = f_{\Pi x^{Prop}.Prop}x$ $\quad B = f_{\Pi x^{Prop}.Type_0}x$

A and B are well-typed terms[3] in $CC_{\omega\,atr}$, under the context Γ:

$$\frac{\Gamma \vdash Prop \rhd Prop : Type_0 \qquad \Gamma, _ : Prop \vdash Prop \rhd Prop : Type_0 \qquad \Gamma \vdash f \rhd f : Prop \to Prop \qquad \Gamma \vdash x \rhd x : Prop}{\Gamma \vdash A \rhd A : Prop}$$

$$\frac{\dfrac{\Gamma \vdash f \rhd f : Prop \to Prop \qquad \Gamma \vdash Prop \to Prop \leq Prop \to Type_0}{\Gamma \vdash f \rhd f : Prop \to Type_0} \qquad \Gamma \vdash x \rhd x : Prop \qquad \cdots}{\Gamma \vdash B \rhd B : Type_0}$$

They also have the same skeleton : $|A| = |B| = f\ x$. However, they are not convertible in $CC_{\omega\,atr}$: if it were the case, by *Confluence*, we could prove that the annotations inside A and B both reduce to a same term P_0:
$\Gamma \vdash \Pi x^{Prop}.Prop \quad \rhd^+ \quad P_0 : s$
$\Gamma \vdash \Pi x^{Prop}.Type_0 \quad \rhd^+ \quad P_0 : t$
By *Generation*, the first reduction entails P_0 to be syntactically equal to $\Pi x^{Prop}.Prop$, and the second one to be syntactically equal to $\Pi x^{Prop}.Type_0$. From these two equations, it is possible to prove that $Prop$ and $Type_0$ are equal, which in contradiction with the definition of sorts. Therefore, *Erased Confluence* does not hold for $CC_{\omega\,atr}$.

□

The issue arises in the annotation process that lifts term from CC_ω to $CC_{\omega\,atr}$, more precisely when applying a variable to a term: in the framework of PTSs, the *Weak shape of types* property was here to force the head of an application to either have a unique type modulo conversion, or to be a λ-abstraction. We made a clever use of this property to show the *Erased Confluence* property of PTS_{atr}. If the head of an application was a variable, we were always sure that its type was "unique" modulo β-conversion.

[3]The following derivations are shortened for the sake of simplicity, the complete typing of annotations is not the important part of this counterexample.

This is no more the case for $CC_{\omega\,atr}$: we can use the SUB rule between the introduction of a variable and the APP rule, like we did in the counterexample, and change the type of the variable for a *strictly* greater one. This possibility adds another conclusion to the second part of the conclusion of *Weak shape of types* (see Lemma 3.3.2), where M can be a variable.

Lemma 5.2.2 (Weak shape of types in $CC_{\omega\,atr}$)**.** *If* $\Gamma \vdash M \rhd ? : A$ *and* $\Gamma \vdash M \rhd ? : B$*, then:*

- *either* $\Gamma \vdash A \cong_\beta B$
- *or we are in one of the following cases:*
    1. *there are* U *and* V *such that* $\Gamma \vdash M \rhd \lambda x^U.V : A$ *and* $\Gamma \vdash M \rhd \lambda x^U.V : B$.
    2. *there is* s *such that* $\Gamma \vdash M \rhd s : A$ *and* $\Gamma \vdash M \rhd s : B$.
    3. *there are* U *and* V *such that* $\Gamma \vdash M \rhd \Pi x^U.V : A$ *and* $\Gamma \vdash M \rhd \Pi x^U.V : B$.
    4. *there is* x *such that* $\Gamma \vdash M \rhd x : A$ *and* $\Gamma \vdash M \rhd x : B$.

This lemma is proved in the same way we proved *Weak shape of types* for PTS_{atr} (see Lemma 3.3.2). Therefore, during the proof of *Erased Confluence*, we are no longer sure, in the difficult case, that the head is always an abstraction: it can be a variable, what prevents us to erase the annotation by reducing a β-redex.

5.2.2 Other attempts and possible leads

In order to avoid the previous trap, we first tried to restrain the possible occurrences of subtyping inside the typing derivations, especially before using the application rule. As said earlier, CC_ω can be modified to become totally *syntax-directed*, which would prevent to insert cumulativity between a VAR and an APP rule. Such a syntax-directed version of $CC_{\omega\,atr}$ enjoys almost all the required properties to make a valid candidate to study, but in fact, it lacks one major property, which is the *Right-Hand Reflexivity* lemma. As we noticed while proving the *Erased Confluence* for PTS_{atr}, without this reflexivity lemma, there is no way we would be able to correctly apply the *Weak shape of type* lemma to prove that the whole annotation process is valid, and

so to validate our candidate intermediate system for the equivalence.

One has to remember the first reason we added those annotations on the applications: we needed a remainder of the type information that were forgotten by an application, the type of the function. Since we can now modify this type by cumulativity rather than equality, we need a way to reflect this behavior into the annotations. This solution was our next attempt, by adding a new rule that explicitly allows to change the sort level of an annotation:

$$\frac{\begin{array}{c}\Gamma \vdash A \rhd A' : s \qquad \Gamma(x:A) \vdash \Pi\Delta \rhd \Pi\Delta' : t \\ \Gamma \vdash M \rhd M' : \Pi x^A.\Delta v \qquad \Gamma \vdash N \rhd N' : A \qquad v < u\end{array}}{\Gamma \vdash M_{\Pi x^A.\Delta u} N \rhd M'_{\Pi x^{A'}.\Delta' v} N' : \Pi\Delta[N/x]} \text{ APP2}$$

Here, the ending sort on the annotation is strictly decreasing from u to v. So the counterexample can be avoided by first making the sort of the annotation go from $Type_0$ to $Prop$, and then making the confluence happen. This time, the system also enjoys the *Right-Hand Reflexivity* property, but it is the *diamond property* which is failing. This APP2 rule easily commutes with APP and itself, but does not behave well with the BETA rule. Let us take a look at this particular case:

1. We are considering two derivations starting form a β-redex:

   $\Gamma \vdash (\lambda x^A.M)_{\Pi x^C.\Delta u} N \rhd M'[N'/x] : \Pi\Delta[N/x]u$ is an instance of BETA.

   $\Gamma \vdash (\lambda x^A.M)_{\Pi x^C.\Delta u} N \rhd (\lambda x^{A''}.M'')_{\Pi x^{C'}.\Delta'' v} N'' : \Pi\Delta[N/x]u$ is an instance of APP2.

2. By induction hypothesis and *Generation*, we can find two ways to type M (with $v < u$):

   (a) $\Gamma(x:A) \vdash M \rhd M' : \Pi\Delta u$

   (b) $\Gamma \vdash \lambda x^A.M \rhd \lambda x^{A''}.M'' : \Pi x^A.\Delta v$

By using *Generation* one more time, we can extract a typing judgment about M from the second one:

$$\Gamma(x:A) \vdash M \rhd M'' : B \text{ and } \Gamma \vdash \Pi x^A.B \leq_\beta \Pi x^A.\Delta v \quad (1)$$

So we can apply the induction hypothesis to find a common reduct M_0 (resp. N_0) for M' and M'' (resp. N' and N''), and build a candidate to close the diamond: $M_0[N_0/x]$. The next step is to prove that the following two judgments are correct:

$$\Gamma \vdash M'[N'/x] \rhd M_0[N_0/x] : \Pi\Delta[N/x]u$$
$$\Gamma \vdash (\lambda x^{A''}.M'')_{\Pi x^{C'}.\Delta''v} N'' \rhd M_0[N_0/x] : \Pi\Delta[N/x]u$$

The first one is easy to prove since this system has the *Substitution* property. However, to prove the second one, we have to apply the BETA rule, which will require the following judgment to hold: $\Gamma(x : A'') \vdash M'' \rhd M_0 : \Pi\Delta''v$. Unfortunately, M_0 is typed by B, not by $\Pi\Delta v$ (which would be enough since Δ reduces to Δ''). At this point, we have no way to relate B and $\Pi\Delta v$ directly, they are hidden inside Π-types in (1).

To be able to prove that we can actually exchange APP2 and BETA, we would need to already have the injectivity of Π-types in order to be able to extract the missing piece of information from (1). Since we are trying to prove the *diamond* property, we do not have this possibility at this point of the development: we are still facing the loop between confluence and injectivity of products.

Our most recent attempts are trying to break this loop by adding explicit marks when using the strict cumulativity of Π-types during the conversion, but this is still too early to enter into the details.

5.3 Other leads for future investigations

As we just saw, there is room for direct improvements of this work, but we would like to explore new paths too, especially with different kinds of logics:

- The parallel between natural deduction and sequent calculus that we saw in Chapter 2 seems to be a good way to explore the connexion between inductive and co-inductive types in the *Calculus of Inductive Constructions*. Besides the categorical duality, it seems there is an alternative syntactic duality between these two kinds of type constructions, and also between the guard conditions that we need to respect if we want to prove that the calculus is terminating. A presentation

inspired by the $\overline{\lambda}\mu\tilde{\mu}$-calculus of Curien and Herbelin [CH00] seems possible.

- We saw that PTSs are a good way to build the meta-theory of type systems and to factorize the work by abstracting some part of the type system. However PTSs have always been looked at through the lens of intuitionistic logic (or classical logic if one wants to add classical axioms). A new interesting approach may be to look at *linear* logics and try to understand what could be a PTS in this framework, and what kinds of new concept can emerge from such constructions. This idea to translate a particular type system into a linear one as already been tried (e.g. the *Celf* [SNS] language which is an extension of the LF framework with linear types), but doing this translation to abstract systems such as PTSs seems worth investigating, to extend our knownledge of linearity with dependent types.

Bibliography

[Abe10] Andreas Abel. Towards normalization by evaluation for the *betaeta*-calculus of constructions. In *FLOPS*, pages 224–239, 2010.

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[ACD07] Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf Type Theory with typed equality judgements. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 3–12. IEEE Computer Society Press, 2007.

[ACP+08] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL*, pages 3–15, 2008.

[Ada06] Robin Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006.

[Alt93] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.

[Bara] Bruno Barras. Coq contribution: A formalisation of Pure Type Systems. http://www.lix.polytechnique.fr/coq/contribs/PTS.html.

[Barb] Bruno Barras. Sets in Coq, Coq in Sets. http://www.lix.polytechnique.fr/~barras/proofs/sets/.

[Bar84] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[Ber88] Stefano Berardi. *Type Dependence and Constructive mathematics*. PhD thesis, Mathematical Institute Torino, 1988.

[BW97] Bruno Barras and Benjamin Werner. Coq in coq. Technical report, 1997.

[CH00] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.

[Chu40] Alonzo Church. A Formulation of the Simple Theory of Types. *J. Symb. Log.*, 5(2):56–68, 1940.

[Chu51] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1951. (second printing, first appeared 1941).

[Coq] Coq Development Team. The Coq Proof Assistant Reference Manual. http://coq.inria.fr/refman/.

[Cor97] Cristina Cornes. *Conception d'un langage de haut niveau de representation de preuves: Récurrence par filtrage de motifs; Unification en prsence de types inductifs primitifs; Synthèse de lemmes d'inversion*. PhD thesis, Université Paris 7, November 1997.

[CP03] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[dB72] N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Fan97] Song Fangmin. Expansion postponement in pure type systems. *J. of Comput. Sci. & Technol.*, 12(6):555–563, 1997.

[FMCM] Amy Felty, Alan Martin, Roy Crole, and Alberto Momigliano. Hybrid: a package for higher-order syntax in isabelle and coq. http://hybrid.dsi.unimi.it/.

[Gal] Équipe Gallium. Le langage Caml. http://caml.inria.fr/.

[Geu93] Herman Geuvers. *Logics and Type Systems.* PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation fonctionelle de Gödel l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, 1971.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theor. Comput. Sci.*, 13:225–230, 1981.

[GR02] Francisco Gutiérrez and Blas C. Ruiz. A cut-free sequent calculus for pure type systems verifying the structural rules of gentzen/kleene. In *LOPSTR*, pages 17–31, 2002.

[GR03] Francisco Gutiérrez and Blas C. Ruiz. Cut elimination in a class of sequent calculi for pure type systems. *Electr. Notes Theor. Comput. Sci.*, 84, 2003.

[GW94] Herman Geuvers and Benjamin Werner. On the church-rosser property for expressive type systems and its consequences for their metatheoretic study. In *LICS*, pages 320–329, 1994.

[HAS] Haskell programming language. http://www.haskell.org/.

[Her94] Hugo Herbelin. A lambda-calculus structure isomorphic to gentzen-style sequent calculus structure. In *CSL*, pages 61–75, 1994.

[Her95] Hugo Herbelin. *Séquents qu'on calcule*. PhD thesis, Université Paris 7, 1995.

[How80] W.A. Howard. *The formulae-as-types notion of construction*, pages 479–490. Academic Press, London, New York, 1980.

[HU10] Brian Huffman and Christian Urban. A New Foundation for Nominal Isabelle. In *ITP*, pages 35–50, 2010.

[Hue75] Gérard P. Huet. Unification in Typed Lambda Calculus. In *Lambda-Calculus and Computer Science Theory*, pages 192–212, 1975.

[Hue02] Gérard Huet. Constructive computation theory. *Course notes on lambda calculus, University of Bordeaux I, 2002.*, 2002. http://yquem.inria.fr/~huet/PUBLIC/CCT.pdf.

[Len06] Stéphane Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory*. PhD thesis, Université Paris 7 & University of St Andrews, 2006.

[Luo89] Z. Luo. ECC: An extended calculus of constructions. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 385–395, Piscataway, NJ, USA, 1989. IEEE Press.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen*, volume 475 of *LNAI*, pages 253–281. Springer-Verlag, 1991.

[Miq01] Alexandre Miquel. *Le calcul des constructions implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, December 2001.

[ML71] Per Martin-Löf. A Theory of Types. 1971.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[MP93] James McKinna and Robert Pollack. Pure Type Systems Formalized. In *TLCA*, pages 289–305, 1993.

[MP99] James McKinna and Robert Pollack. Some Lambda Calculus and Type Theory Formalized. *J. Autom. Reasoning*, 23(3-4):373–409, 1999.

[MW97] P.-A. Melliès and B. Werner. A generic normalization proof for pure type systems. In C. Paulin-Mohring and E. Gimenez, editors, *TYPES'96*. LNCS, Springer-Verlag, 1997.

[NPW] Tobias Nipkow, Larry Paulson, and Makarius Wenzel. Isabelle Proof Assistant. http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pol92] R. Pollack. Typechecking in Pure Type Systems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, June 1992.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Univ. of Edinburgh, 1994.

[Pol98] Erik Poll. Expansion postponement for normalising pure type systems. *J. Funct. Program.*, 8(1):89–96, 1998.

[POP] POPLmark Challenge. http://alliance.seas.upenn.edu/~plclub/cgi-bin/poplmark/index.php?title=The_POPLmark_Challenge.

[PS] Frank Pfenning and Carsten Schürmann. The twelf project. twelf.plparty.org/.

[Sch24] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen 92*, pages 305–316, 1924.

[SH10] Vincent Siles and Hugo Herbelin. Equality is typable in semi-full pure type systems. In *Proceedings, 25th Annual IEEE Symposium on Logic in Computer Science (LICS '10), Edinburgh, UK, 11-14 July 2010*. IEEE Computer Society Press, 2010.

[Sila] Vincent Siles. Formalization of equivalence between PTS and PTSe. http://www.lix.polytechnique.fr/~vsiles/coq/PTSATR.html.

[Silb] Vincent Siles. Formalization of sequent calculus pure type system. http://www.lix.polytechnique.fr/~vsiles/coq/formalisation.html.

[SML] Standard ML of New Jersey. http://www.smlnj.org/.

[SNS] Anders Schack-Nielsen and Carsten Schürmann. The Celf language. http://www.logosphere.org/~celf/pmwiki/index.php.

[Str91] Thomas Streicher. *Semantics of type theory: correctness, completeness, and independence results*. Birkhauser Boston Inc., Cambridge, MA, USA, 1991.

[Tak95] Masako Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, 118(1):120–127, 1995.

[Urb08] Christian Urban. Nominal Techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008. http://isabelle.in.tum.de/nominal/example.html.

[vBJ93] L. S. van Benthem Jutting. Typing in pure type systems. *Inf. Comput.*, 105(1):30–41, 1993.

[vBJMP93] L. S. van Benthem Jutting, James McKinna, and Robert Pollack. Checking algorithms for pure type systems. In *TYPES*, pages 19–61, 1993.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[WL] Benjamin Werner and Gyesik Lee. A simple model of calculus of inductive constructions with judgemental equality. unpublished manuscript.

[WR27] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.