Generic Bidirectional Typing in a Logical Framework for Dependent Type Theories

Thiago Felicissimo

Contents

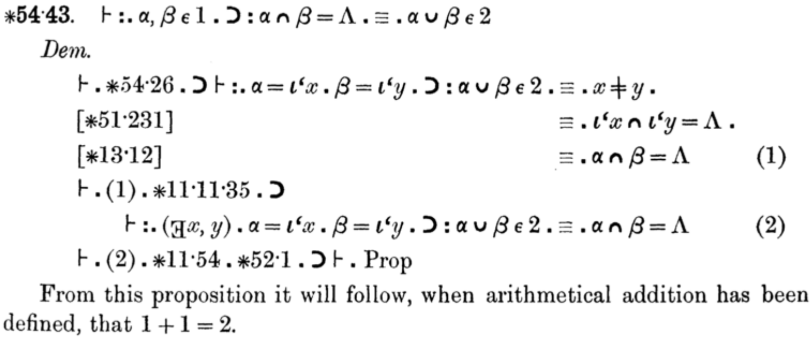
Chapter 1

Introduction

From formal mathematics to mechanized proofs

The birth of *modern mathematical logic* in the turning of the 20th century is rooted in the realization that, unlike the mathematical practice until then, mathematics should be developed in a *formal system*, in which statements are derived from a fixed collection of axioms and deduction rules. Central in this vision is that, unlike earlier attempts at formalizing mathematics — such as that of Euclid — mathematical objects are treated abstractly and striped of any intuitive meaning not implied by the axioms.

Yet, while mathematical rigor has improved, the modern practice has still remained mostly informal. The reason for this is quite mundane: for most of the last century, no method for building formal proofs was available other than manually writing them in painstaking detail. An example of this can be found in the following well-known excerpt from Whitehead and Russell's *Principia Mathematica*, which illustrates the impracticality of this method.



$$
\begin{array}{lll}
\textbf{*54·43}. \quad \vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2 & & \\
\quad \textit{Dem.} & & \\
\qquad \vdash . *54{\cdot}26 . \supset \vdash :. \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y . & & \\
\qquad [*51{\cdot}231] & \equiv . \iota`x \cap \iota`y = \Lambda . & \\
\qquad [*13{\cdot}12] & \equiv . \alpha \cap \beta = \Lambda & (1) \\
\qquad \vdash . (1) . *11{\cdot}11{\cdot}35 . \supset & & \\
\qquad\qquad \vdash :. (\exists x, y) . \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda & & (2) \\
\qquad \vdash . (2) . *11{\cdot}54 . *52{\cdot}1 . \supset \vdash . \text{Prop} & &
\end{array}
$$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

Fortunately, this situation changed in the second half of the 20th century with the advent of computers, making formal mathematics practically possible, and starting a fruitful collaboration between logic and computer science. Indeed, while verifying thousands of applications of deduction rules would take considerable time and effort for a mathematician, the same task can now be performed automatically by a machine.

Moreover, while no algorithm can prove by itself arbitrary statements in sufficiently expressive logics, modern *proof assistants* can still provide help with the writing of proofs. First, low-level proof steps may be grouped into more readable commands closer to informal mathematical practice, called *tactics*. Second, tedious and uninteresting steps, usually omitted from informal proofs, can sometimes be filled automatically with the use of automatic provers, such as SMT solvers. Interestingly, the help provided by proof assistants can even allow for proofs that would be infeasible to write by hand, such as that of the Four Color Theorem, which requires the analysis of hundreds of configurations [G⁺08]. All in all, modern proof systems have allowed for the formalization of a significant corpus of mathematical knowledge.

Type theories

While set theory is the most well-accepted foundation in informal mathematical practice, the situation in the world of mechanized mathematics is quite different. Modern proof systems are mostly based on *type theories*, formal systems in which objects are not treated in a monolithic way, but classified according to their *types*. One notable example of such theory, Church's *Simple Theory of Types (STT)*[1] [Chu40, BA24], is the basis of various proof systems such as HOL-Light, HOL4 and Isabelle/HOL, the latter being among the most popular proof assistants today.

Compared with set theory, an interesting aspect of Church's STT is its uniform treatment of propositions and objects as just λ-terms of certain types. Yet, the concept of proof is still given a second-class treatment, like in set theory. One of the key realizations of the 1960s is that this is not necessary: proofs can be internalized as terms, in which case the propositions they prove become themselves types. This *propositions as types* correspondence, also known as the *Curry-Howard isomorphism* [SU06], is at the heart of a second family of type theories, called *dependent type theories* [ML84, dB94, Hof97]. Indeed, for sufficiently expressive logics, capable of talking about objects, the internalization of propositions as types leads to *dependent types*, that is, types which depend on terms. For instance, the proposition "n is prime" now becomes the dependent type whose terms witness the primality of n. Dependent type theories are the foundation of various popular proof assistants, such as Coq, Agda and Lean.

Constructivism and programming

An interesting aspect of dependent type theories is that some of them, referred to as *constructive*, can also be seen as programming languages, providing a unified foundation for both proving and programming. Indeed, just like programming languages, in such

[1] Not to be confused with Church's simply typed λ-calculus, the framework used to specify the syntax of simple type theory, but which is devoid of any notion of provability. Church's Simple Theory of Types is sometimes also referred to as *Higher-Order Logic (HOL)*.

theories a term of boolean type always computes to a *canonical value* of boolean type, namely either true or false, a property known as *canonicity*.

The rationale for calling such theories constructive dates back to the seminal work of Martin-Löf [ML84, ML75] — who instead favored the equivalent terminology of *intuitionistic* — and is justified by the fact that, through the propositions-as-types correspondence, the canonicity property yields exactly the constructive understanding of propositions. Indeed, differently from classical logic, in which propositions are thought of as truth values, constructive logic promotes the idea that propositions should be determined by their *canonical proofs*. For instance, constructive logic is designed in such a way that any proof of $A_1 \vee A_2$ yields a *canonical proof* of $A_1 \vee A_2$, namely either a proof of A_1 or a proof of A_2. Crucial for this to work is the rejection of the *law of excluded middle (LEM)*, stating that $P \vee \neg P$ holds for any P.

Nevertheless, while constructivism is one of the main qualities sought by designers of dependent type theories, it is worth noting that, when dropping this condition, type theory becomes able to subsume many formal systems of interest. For instance, by internalizing proofs as terms in Church's STT, one roughly obtains an extension of the Calculus of Constructions (CoC) [CH88] — the dependent type theory at the origin of the Coq proof assistant — with non-constructive principles.[2] These added principles disturb the canonicity property of CoC, making the resulting theory non-constructive, but allow us to see dependent type theories as a subsuming formalism for many formal systems. For this reason, dependent type theories will be the main formal systems of interest in this thesis. Accordingly, we will not consider constructivism as a necessary quality of such theories, but rather one that should be sought whenever possible.

Logical Frameworks (LFs)

The full definition of dependent type theories usually requires one to deal with various syntactic bureaucracies. Most frustratingly, a great deal of this work is dedicated to establishing universal notions that ideally should be defined once and for all, such as *variable binding* and *substitution*. A *Logical Framework (LF)* [HHP93, Pfe01b, Har21b] addresses this fact by proposing a unified meta-language for defining theories, in which such universal notions are built in by default. As such, when defining an object-theory[3] in a logical framework, the mathematician is freed from most of the aforementioned bureaucracies, and can concentrate on the logical aspects specific to the theory in question.

Moreover, by providing a setting in which universal notions are represented in a common way, logical frameworks allow for establishing basic metatheorems, usually shown in a case-by-case basis, once and for all, effectively allowing for a *unified theory of*

[2] The precise relationship is however a bit more subtle; the reader is referred to Geuvers [Geu95] for more details.

[3] We refer to the theories defined inside an LF as *object-theories* or *object-languages*, as opposed to the meta-language, which is the framework.

type theories. For instance, most LFs are designed specifically for *structural* theories, and so satisfy the *weakening* property, stating that judgment derivability is monotone with respect to context extension. Any theory formulated in such frameworks thus satisfies weakening automatically.

Finally, by providing a setting in which various theories can be defined uniformly, LFs allow for comparing their logical features in a more direct way. Indeed, while theories might differ in unessential ways when presented with distinct syntactic conventions — such as using either single or simultaneous substitution, or defining variables either as de Bruijn indices or using names — these differences vanish inside an LF, allowing only logical aspects to be highlighted.

The Edinburgh Logical Framework

While the ideas that led to the invention of LFs can be traced back to de Bruijn's AUTOMATH [dB94] and to Martin-Löf himself [ML84, NPS90], the first LF to be explicitly named as such was the *Edinburgh Logical Framework (ELF)* [HHP93], constituting of a minimalistic dependent type theory. In the ELF methodology [HL07, Pfe01a], one usually starts by defining the raw syntax of the object-theory with the use of *Higher-Order Abstract Syntax (HOAS)* [PE88], a technique that has its roots in Church's STT. For instance, the following ELF declarations specify the raw syntax of a basic dependent type theory:

$$
\begin{aligned}
&\mathsf{ty}, \mathsf{tm} : \mathsf{Type}\\
&\Pi : \mathsf{ty} \longrightarrow (\mathsf{tm} \longrightarrow \mathsf{ty}) \longrightarrow \mathsf{ty}\\
&\lambda : (\mathsf{tm} \longrightarrow \mathsf{tm}) \longrightarrow \mathsf{tm}\\
&@ : \mathsf{tm} \longrightarrow \mathsf{tm} \longrightarrow \mathsf{tm}
\end{aligned}
$$

While the type-level symbols ty and tm represent syntactic classes, the term-level symbols Π, λ and $@$ represent syntactic constructors of the object-theory's syntax. We can for instance represent the object term $(\lambda x.t)y$ as the framework term $@\ (\lambda\ (x.[\![t]\!]))\ y$ of framework type tm — writing $x.t$ for the framework's abstraction, and $[\![-]\!]$ for the function mapping an object term to its representation in the framework. Two main benefits of defining theories in ELF can already be seen. First, variables of the framework are used to represent variables of the object-theory, which therefore never need to be defined explicitly. Second, the framework's function type is not only used to specify how many arguments each symbol expects but also to specify variable binding, which is also a primitive notion covered once and for all by the framework.

Judgments-as-types

With the raw syntax of a type theory defined, its judgments can then be represented with the *judgments-as-types* principle. For instance, in dependent type theory one usually has four basic judgment forms: A type and $A \equiv B$ type and $t : A$ and $t \equiv u : A$. These can be represented in ELF by declaring the following type-level symbols.

$$\begin{aligned}
\mathrm{Ty} &: \mathrm{ty} \longrightarrow \mathsf{Type} \\
\mathrm{Ty}^{\equiv} &: \mathrm{ty} \longrightarrow \mathrm{ty} \longrightarrow \mathsf{Type} \\
\mathrm{Tm} &: \mathrm{tm} \longrightarrow \mathrm{ty} \longrightarrow \mathsf{Type} \\
\mathrm{Tm}^{\equiv} &: \mathrm{tm} \longrightarrow \mathrm{tm} \longrightarrow \mathrm{ty} \longrightarrow \mathsf{Type}
\end{aligned}$$

Of course, the above basic judgment forms should be generalized to *hypothetical judgment forms*, but once again this is covered automatically by the framework. For instance, the judgment form $t : A$ in an hypothetical object-context $\Gamma = y : B$ is represented as the framework type $\mathrm{Tm}\ [\![t]\!]\ [\![A]\!]$ in the framework context $[\![\Gamma]\!] = y : \mathrm{tm}, y' : \mathrm{Tm}\ y\ [\![B]\!]$.

Object-theory derivation rules can then be represented in the framework using term-level symbols. For instance, the rules

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B \text{ type} \atop \Gamma \vdash t : \Pi x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[u/x]} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B \text{ type}}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash t \equiv u : A}{\Gamma \vdash u \equiv t : A}$$

are represented in ELF by the following symbol declarations:[4]

$$\begin{aligned}
\mathrm{Tm}/@ :\ & \mathrm{Ty}\ \mathsf{A} \longrightarrow ((x : \mathrm{tm}) \longrightarrow \mathrm{Tm}\ x\ \mathsf{A} \longrightarrow \mathrm{Ty}\ (\mathsf{B}\ x)) \longrightarrow \\
& \mathrm{Tm}\ \mathsf{t}\ (\Pi\ \mathsf{A}\ \mathsf{B}) \longrightarrow \mathrm{Tm}\ \mathsf{u}\ \mathsf{A} \longrightarrow \mathrm{Tm}\ (@\ \mathsf{t}\ \mathsf{u})\ (\mathsf{B}\ \mathsf{u}) \\
\mathrm{Tm}/\mathrm{conv} :\ & \mathrm{Tm}\ \mathsf{t}\ \mathsf{A} \longrightarrow \mathrm{Ty}^{\equiv}\ \mathsf{A}\ \mathsf{B} \longrightarrow \mathrm{Tm}\ \mathsf{t}\ \mathsf{B} \\
\mathrm{Tm}^{\equiv}/\mathrm{sym} :\ & \mathrm{Tm}^{\equiv}\ \mathsf{t}\ \mathsf{u}\ \mathsf{A} \longrightarrow \mathrm{Tm}^{\equiv}\ \mathsf{u}\ \mathsf{t}\ \mathsf{A}
\end{aligned}$$

Note how the operation of substitution in the first rule is covered automatically by the framework using the application $\mathsf{B}\ \mathsf{u}$. Indeed, when B and u are instantiated with terms $x.B$ and u the result becomes convertible to $B[u/x]$ by the framework-level β-equality. Note also that there is no need to declare a symbol to represent the variable rule, which is handled by the framework's own variable rule.

From derivations-as-terms to subjects-as-terms

By declaring term-level symbols to represent derivation rules, the derivations of a judgment $t : A$ are represented as framework terms of type $\mathrm{Tm}\ [\![t]\!]\ [\![A]\!]$ — because of this, the motto judgments-as-types is often followed by *derivations-as-terms*. The framework typing judgment $u : \mathrm{Tm}\ [\![t]\!]\ [\![A]\!]$ then asserts that the term u represents an object-theory derivation of $t : A$. This representation is however not accurate with respect to the

[4] Following ELF notational conventions, we omit from the types the arguments that can be inferred from the rest.

meaning of this object-theory judgment, which instead asserts that t, the *subject* of the judgment, is a term of type A. Indeed, not only the distinction between the subject and the other constituents is dropped, but one also represents the judgment *synthetically* — meaning requiring evidence — instead of *analytically* — meaning self-evident — as argued by Harper [ML94, Har21b]. Moreover, as pointed out by Sterling [Ste22b], the semantics of such presentations does not agree with the usual semantics of the represented type theory: instead of a set of types and a type-indexed family of terms, one instead gets two sets of raw terms and types along with families of derivations of their typing judgments. For this reason, we say that the above methodology does not yield a direct definition of the represented theory, but instead only of its *deductive machinery*.

In order to define type theories directly we can apply a different principle, which we dub *judgment-as-types, subjects-as-terms*.[5] The reason for this name is that we now distinguish the subject from other constituents of a judgment: types now represent judgments *without* subjects, which are instead represented as their inhabitants. For instance, by declaring

$$\begin{aligned}
&\mathrm{Ty} : \mathsf{Type}\\
&\mathrm{Tm} : \mathrm{Ty} \to \mathsf{Type}\\
&\Pi : (\mathsf{A} : \mathrm{Ty}) \to (\mathsf{B} : \mathrm{Tm}\ \mathsf{A} \to \mathrm{Ty}) \to \mathrm{Ty}\\
&\lambda : (\mathsf{t} : (x : \mathrm{Tm}\ \mathsf{A}) \to \mathrm{Tm}\ (\mathsf{B}\ x)) \to \mathrm{Tm}\ (\Pi\ \mathsf{A}\ \mathsf{B})\\
&@ : (\mathsf{t} : \mathrm{Tm}\ (\Pi\ \mathsf{A}\ \mathsf{B})) \to (\mathsf{u} : \mathrm{Tm}\ \mathsf{A}) \to \mathrm{Tm}\ (\mathsf{B}\ \mathsf{u})
\end{aligned}$$

we can represent the object-theory judgment $t : A$ as the framework judgment $[\![t]\!] : \mathrm{Tm}\ [\![A]\!]$, that now correctly captures the intended meaning.

In the above approach, it is also worth noting that no symbols need to be declared to represent object-theory equality judgments, which are instead represented by the framework's own equality judgment: for instance, $t \equiv u : A$ is represented as just $[\![t]\!] \equiv [\![u]\!] : \mathrm{Tm}\ [\![A]\!]$. This also means that we do not need to declare any symbol to represent the object-theory conversion rule, which is covered automatically by the framework's conversion rule.

From pure LFs to equational LFs

If we intended to define a type theory with dependent functions, the above declarations are incomplete. Indeed, we still need to ensure that the terms[6] $@\ (\lambda\ (x.t))\ u$ and $t[u/x]$ are convertible in order to correctly represent the object-theory β-equality. However, ELF is an example of a *pure LF*, meaning it only allows for the declaration of generators (that

[5] Unfortunately, the literature uses the name judgments-as-types interchangeably to designate the two principles. Because of this, we have chosen to name the second one with the subjects-as-terms suffix to distinguish it from the first. The obtained presentations are also respectively called *synthetic* and *analytic* by Harper [Har21b]. Finally, it is worth mentioning that hybrid approaches are also possible, and not all encodings can be classified as applying only one of these two principles.

[6] Omitting the domain and co-domain annotations of $@$ and λ, for readability purposes.

is, symbols) but no equations. Therefore, in ELF it is impossible to directly define type theories whose definitional equality is non-trivial.

The aforementioned problem can be addressed by moving to an *equational LF* [Har21a, Car86, NPS90, Uem21, Ada08], in which one is also allowed to constrain the terms of the theory with equations. In this setting, we can finish the definition of the above theory by declaring the equation (once again, omitting the domain and co-domain annotations).

$$@\ (\lambda\ \mathsf{t})\ \mathsf{u} \equiv \mathsf{t}\ \mathsf{u}$$

The above discussion might give the impression that the lack of support for equations in pure LFs is a deficiency. Actually, this is a careful design choice, for it ensures that equality and typing are always decidable in such frameworks, enabling for instance the implementation of ELF in tools like Twelf [GPS99] and Beluga [PD10]. Moreover, because object-theory derivations can be represented as terms in the framework, such implementations can be used for mechanizing the metatheory of type theories and programming languages in a concise way [HL07, LCH07], without the need for formalizing basic notions such as binding and substitution. On the other hand, for this same reason, one does not obtain from such implementations a type-checker for the object-theory, but rather for its typing derivations.

By moving to an equational LF, the above problem is apparently solved: now, an object-theory judgment $t : A$ can be directly represented as a framework judgment $[\![t]\!] : \mathrm{Tm}\ [\![A]\!]$, and so type-checking in the object-theory is reduced to type-checking in the framework. However, because such frameworks allow users to pose arbitrary equations, it is not true anymore that type-checking is always decidable, and one would instead need to design specific equality-checking algorithms for each theory. For this reason, while equational LFs have been around since Cartmell's seminal work in the 1980s [Car86], their implementations have been comparatively much less studied than those of pure LFs.

Dedukti: An implementable equational LF

If allowing arbitrary equations immediately breaks decidability of equality in the framework, a possible solution to this is to instead restrict those which are supported. This is exactly the approach taken by the Dedukti logical framework [CD07, Sai15b, BDG+23], an equational LF in which the only accepted equations are those generated by *untyped rewrite rules* [BKdVT03]. This means that equations have a preferred direction of application and are defined without making any reference to typing.[7]

While this restricts the class of theories one can define, it has the advantage of allowing one to decide the equality theory in a uniform way. Indeed, computing normal forms and comparing them is always a sound equality-checking procedure, which moreover becomes

[7] Therefore, unlike the LFs discussed before, equality in Dedukti is not a proper judgment but instead an external relation defined on raw terms, henceforth called *conversion*.

complete whenever the set of rewrite rules is sufficiently well behaved. The fact that equality can be (semi-)decided in a uniform way is the key property which has allowed for the implementation of Dedukti in tools like DkCheck [Sai15b] and LambdaPi [HB20].

What are then the benefits of having an implementable equational LF? First, while representing theories in ELF yielded a type-checker for derivations, representing theories in Dedukti yields proper type-checkers for their terms. In order words, Dedukti provides a *generic* type-checker, which can be used as an independent verifier for proofs coming from proof assistants and automated provers, improving trust in their correctness. Moreover, by representing multiple type theories in a common setting, Dedukti also strives to be a unifying framework for defining proof transformations, enabling the sharing of proofs between various proof systems — this point will be further discussed in the introduction of Part III.

Finally, it is worth mentioning that while Dedukti was, for some time, the only equational LF developed with practical applications in mind, this has recently changed with the advent of the Andromeda project, implementing the framework of *Finitary Type Theories (FTTs)* [HB23]. Compared with Dedukti, Andromeda allows for arbitrary equations, and so it does not aspire to provide a provably complete equality-checking algorithm. Instead, it relies mostly on a system of user-defined *handlers* which try to check equality automatically. Moreover, for equations that can be classified as either computation or extensionality rules, it also provides a built-in checker that, while not being provably complete, is sound and works similarly to other algorithms used in practice [BK22].

This thesis

The overarching goal of this thesis is to advance the study of equational LFs as practical tools, particularly in the context of the Dedukti project. For this, we propose three main contributions, each being the subject of one of the parts of this thesis:[8]

1. We start in Part I by revisiting the problem of showing that object-theories are correctly represented in the logical framework. More precisely, it is a recurring problem in equational LFs that the *conservativity* of therein defined theories with respect to their usual presentations is not always evident [Hof97]. We propose a revised methodology for Dedukti encodings which allows for showing conservativity easily, by banning so-called *arrow-producing* rewrite rules, which have historically been heavily used in the Dedukti literature [BDG+23]. The cornerstone of our proposal is a new normalization criterion for β-reduction in Dedukti that, provided no arrow-producing rules are used, allows one to consider only β-normal forms when proving conservativity, which can then be shown by a simple induction. We illustrate our methodology by proposing a new encoding of functional Pure Type

[8] The title of this thesis refers specifically to the second part, as it is the longest and its contribution is the most substantial.

Systems in Dedukti that, unlike the previously proposed encoding [CD07], can easily be shown conservative.

2. In Part II we contribute a new logical framework which, compared with Dedukti, allows for representing type theories with their usual non-annotated syntaxes. Indeed, a deficiency of frameworks like Dedukti is that theories are presented in a *fully annotated* manner, in which all arguments are spelled out explicitly, worsening the user experience and the performance of type-checking. The key to allowing for the removal of these annotations without jeopardizing decidability of typing is the integration of *bidirectional typing*, a discipline in which the typing judgment is decomposed into *inferring* and *checking* modes. While bidirectional typing has been well known in the literature for quite some time, one of the contributions of our work is that, by formulating it in a logical framework, we are able to give it a *generic* treatment for all theories fitting our framework. Our proposal has been implemented in the generic type-checker BiTTs, allowing it to be used in practice with various theories.

3. Finally, we demonstrate in Part III how Dedukti can be used in practice as a tool for translating proofs, by proposing a proof transformation for sharing proofs with *predicative* systems. This transformation has in particular allowed for the translation of proofs from Matita to Agda, two proof assistants based respectively on *impredicative* and *predicative* type theory, yielding the first ever proofs in Agda of Fermat's Little Theorem and of Bertrand's Postulate. The definition of this transformation also required us to study the theory of equational unification for *predicative universe levels*, a subject that until now had been overlooked.

We describe the three contributions very succinctly because each of this thesis' parts starts with a high-level introduction, which the reader is invited to refer to for more details.

List of publications

We list the (published or accepted) formal publications written during this thesis:

1. **Adequate and Computational Encodings in the Logical Framework Dedukti**. Thiago Felicissimo. In proceedings of the *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).*

2. **Translating Proofs from an Impredicative Type System to a Predicative One**. Thiago Felicissimo, Frédéric Blanqui and Ashish Kumar Barnawal. In proceedings of the *31st EACSL Annual Conference on Computer Science Logic (CSL 2023).*

3. **Generic Bidirectional Typing for Dependent Type Theories**. Thiago Felicissimo. In proceedings of the *33rd European Symposium on Programming (ESOP 2024).*

4. **Sharing Proofs with Predicative Theories Through Universe Polymorphic Elaboration**. Thiago Felicissimo and Frédéric Blanqui. To appear in *Logical Methods in Computer Science*.

5. **Impredicativity, Cumulativity and Product Covariance in the Logical Framework Dedukti**. Thiago Felicissimo and Théo Winterhalter. To appear in proceedings of the *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*.

The first publication is the subject of Part I, the third is the subject of Part II, the second and the fourth are the subject of Part III, and the last publication is not discussed in this thesis.

Chapter 2

Preliminaries: Dedukti

In the introduction we have motivated and introduced the logical framework Dedukti informally. This preliminary chapter now defines it formally, as it will be a central object of study in this thesis. Because there are many other good introductions to Dedukti [Sai15b, BDG+23], in this chapter we will be rather concise when defining it — when presenting a new logical framework in Part II we will explain it in a more gradual and detailed fashion. Finally, because there are many variations on the definition of Dedukti in the literature, we then conclude this chapter with a discussion about them.

2.1 Dedukti

Given a set of *symbols* $\mathcal{F}$, whose elements we refer to by $f, g, \ldots$, and an infinite set of variables $\mathcal{V}$, whose elements we refer to by $x, y, z, \ldots$ or type-writer characters $\mathtt{x}, \mathtt{A}, \mathtt{t}, \ldots$, we define the raw syntax of Dedukti by the grammars of Figure 2.1. We adopt the convention of writing symbol names in blue.

We call $(x : T) \to U$ a *dependent function type* (also called *dependent product*), $x.t$ an *abstraction* and $t\ u$ an *application*. The terms Type and Kind are often abbreviated using

$$
\begin{array}{rrrl}
 & & s, s' ::= & \mathsf{Type} \mid \mathsf{Kind} \\
\boxed{\mathsf{Tm}} & \ni & t, u, v, T, U ::= & x \mid f \mid s \mid (x : T) \to U \mid x.t \mid t\ u \\
\boxed{\mathsf{Ctx}} & \ni & \Gamma, \Delta ::= & \cdot \mid \Gamma, x : T \\
\boxed{\mathsf{Thy}} & \ni & \mathbb{T} ::= & \cdot \mid \mathbb{T}, f : T \mid \mathbb{T}, l \longmapsto r \qquad \text{with } l \longmapsto r \text{ a rewrite rule}
\end{array}
$$

Figure 2.1: Raw syntax of Dedukti

$\boxed{\Gamma \vdash}$

$$\textsc{EmptyCtx}\quad \frac{}{\cdot \vdash} \qquad\qquad \textsc{ExtCtx}\quad \frac{\Gamma \vdash T : \mathsf{Type}}{\Gamma, x : T \vdash}$$

$\boxed{\Gamma \vdash t : T}$

$$\textsc{Var}\quad x : T \in \Gamma\,\frac{\Gamma \vdash}{\Gamma \vdash x : T} \qquad \textsc{Sym}\quad f : T \in \mathbb{T}\,\frac{\Gamma \vdash}{\Gamma \vdash f : T} \qquad \textsc{Type}\quad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}}$$

$$\textsc{Conv}\quad T \equiv U\,\frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s}{\Gamma \vdash t : U} \qquad \textsc{Pi}\quad \frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, x : T \vdash U : s}{\Gamma \vdash (x : T) \to U : s}$$

$$\textsc{Abs}\quad \frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, x : T \vdash U : s \qquad \Gamma, x : T \vdash t : U}{\Gamma \vdash x.t : (x : T) \to U} \qquad \textsc{App}\quad \frac{\Gamma \vdash t : (x : T) \to U \qquad \Gamma \vdash u : T}{\Gamma \vdash t\ u : U[u/x]}$$

Figure 2.2: Typing rules of Dedukti

the letter s and are used for typing the types of the framework[1].

A *substitution* θ is a finite set of pairs t/x, and we write $u[\theta]$ or $\Gamma[\theta]$ for its application to a term u or context Γ. We write $\mathsf{fv}(t)$ for the *free variables* of t, and $\mathsf{dom}(\theta)$ for the *domain* of θ, defined by $x \in \mathsf{dom}(\theta)$ iff $t/x \in \theta$ for some t.[2]

A *rewrite system* $\mathcal{R}$ is a set of *rewrite rules*, which are pairs of the form $t \longmapsto u$ with t of the form $f\ t_1 \ldots t_k$ and $\mathsf{fv}(u) \subseteq \mathsf{fv}(t)$. We then write $\longrightarrow_{\mathcal{R}}$ for the closure under context and substitution of $\mathcal{R}$, and $\longrightarrow_{\beta\mathcal{R}}$ for $\longrightarrow_{\beta} \cup \longrightarrow_{\mathcal{R}}$ where $\longrightarrow_{\beta}$ is the usual β-reduction. We write $\longrightarrow^{*}_{\beta\mathcal{R}}$ for the reflexive-transitive closure of $\longrightarrow_{\beta\mathcal{R}}$, and $\equiv_{\beta\mathcal{R}}$ for its reflexive-symmetric-transitive closure, usually called *conversion* or *definitional equality*. Most of the time, $\mathcal{R}$ is clear from the context, allowing us to write just $\longrightarrow$ for $\longrightarrow_{\beta\mathcal{R}}$ and $\equiv$ for $\equiv_{\beta\mathcal{R}}$.

A central notion in Dedukti is that of a *theory* $\mathbb{T}$, which is defined in Figure 2.1. Given a theory $\mathbb{T}$, we define the typing rules of Dedukti as the ones of Figure 2.2. Here, the conversion $\equiv$ is the one generated by $\beta\mathcal{R}_{\mathbb{T}}$, where $\mathcal{R}_{\mathbb{T}}$ is the underlying rewrite system of $\mathbb{T}$. Whenever the theory is not clear from the context, we write $\mathbb{T} \triangleright \Gamma \vdash t : T$ instead of $\Gamma \vdash t : T$.

[1] They are usually called *sorts* in the literature, though we will avoid this terminology to prevent a name clash with another unrelated concept also called sort, introduced in Chapter 7.

[2] Some authors define $\mathsf{dom}(\theta)$ as the set of variables for which $x[\theta] \neq x$, however this definition suffers from the drawback of not being invariant under post-composition, a somewhat counter-intuitive property.

$\boxed{\mathbb{T} \vdash}$

$$\frac{}{\cdot \vdash} \qquad\qquad \frac{\mathbb{T} \vdash \qquad \mathbb{T} \triangleright \cdot \vdash T : s}{\mathbb{T}, f : T \vdash} \qquad\qquad \frac{\mathbb{T} \vdash}{\mathbb{T}, l \longmapsto r \vdash}$$

Figure 2.3: Well-typed Dedukti theories

A Dedukti theory $\mathbb{T}$ is said to be *well-typed* when we can derive $\mathbb{T} \vdash$ using the rules of Figure 2.3. This condition imposes no constraints on the rewrite rules, so in most situations we will also ask $\beta\mathcal{R}_{\mathbb{T}}$ to be *confluent*, meaning that $t' \;{}^*\!\longleftarrow\; t \longrightarrow^* t''$ implies $t' \longrightarrow^* t''' \;{}^*\!\longleftarrow\; t''$ for some t''', and to satisfy *subject reduction*, meaning that $\mathbb{T} \triangleright \Gamma \vdash t : T$ and $t \longrightarrow t'$ imply $\mathbb{T} \triangleright \Gamma \vdash t' : T$.

We recall the following basic metaproperties of Dedukti.

Proposition 2.1 (Basic metaproperties). *Let us write $\Gamma \sqsubseteq \Gamma'$ when Γ is a subsequence of Γ'.*

Weakening *Suppose $\Gamma \sqsubseteq \Gamma'$ and $\mathbb{T} \triangleright \Gamma' \vdash$. Then $\mathbb{T} \triangleright \Gamma \vdash t : T$ implies $\mathbb{T} \triangleright \Gamma' \vdash t : T$.*

Substitution property *If $\mathbb{T} \triangleright \Gamma, x : U, \Gamma' \vdash t : T$ and $\mathbb{T} \triangleright \Gamma \vdash u : U$ then $\mathbb{T} \triangleright \Gamma, \Gamma'[u/x] \vdash t[u/x] : T[u/x]$.*

Conversion in context *If $\mathbb{T} \triangleright \Gamma, x : U, \Gamma' \vdash t : T$ and $\mathbb{T} \triangleright \Gamma \vdash U' : s$ and $U \equiv U'$ then $\mathbb{T} \triangleright \Gamma, x : U', \Gamma' \vdash t : T$.*

In the following points, suppose that $\mathbb{T}$ is well-typed.

Validity *If $\mathbb{T} \triangleright \Gamma \vdash t : T$ then either $T = \mathsf{Kind}$ or $\mathbb{T} \triangleright \Gamma \vdash T : s$ for $s = \mathsf{Type}$ or Kind.*

We say that a rule $l \longmapsto r$ preserves typing in $\mathbb{T}$ whenever $\mathbb{T} \triangleright \Gamma \vdash l[\theta] : T$ implies $\mathbb{T} \triangleright \Gamma \vdash r[\theta] : T$, for every θ, Γ, T.

Subject reduction for $\mathcal{R}_{\mathbb{T}}$ *If every rule in $\mathcal{R}_{\mathbb{T}}$ preserves typing in $\mathbb{T}$, then $\mathbb{T} \triangleright \Gamma \vdash t : T$ and $t \longrightarrow_{\mathcal{R}_{\mathbb{T}}} t'$ implies $\mathbb{T} \triangleright \Gamma \vdash t' : T$.*

In the following points, suppose that injectivity of dependent products holds in $\mathbb{T}$, meaning that $(x : T) \to U \equiv (x : T') \to U'$ implies $T \equiv T'$ and $U \equiv U'$.

Subject reduction for β *If $\mathbb{T} \triangleright \Gamma \vdash t : T$ and $t \longrightarrow_{\beta} t'$ then $\mathbb{T} \triangleright \Gamma \vdash t' : T$.*

Inversion of symbol applications *If $f : (x_1 : T_1) \to \cdots \to (x_k : T_k) \to U \in \mathbb{T}$ and $\mathbb{T} \triangleright \Gamma \vdash f t_1 \ldots t_k : U'$, then by defining $\theta_i := \{t_j/x_j\}_{j<i}$ we have $U' \equiv U[\theta_{k+1}]$ and $\mathbb{T} \triangleright \Gamma \vdash t_i : T_i[\theta_i]$ for all $i = 1, \ldots, k$.*

Proof. We refer to Blanqui [Bla01] and Saillard [Sai15b] for detailed proofs — even if there the definition of the typing system is not exactly the same, the proofs for the variant used here are straightforward adaptions of their proofs. ■

2.2 Variations on the definition

We conclude this chapter by discussing some possible variations in the definition of the framework. For readers not very familiar with Dedukti, we suggest to skip this more technical section in a first read.

Type annotations in abstractions

In this work we consider a definition of Dedukti with non-annotated abstractions $x.t$, yet most variants in the literature consider domain-annotated abstractions $x^T.t$ [Fer21, BDG+23, CD07]. The justification often given for this extra annotation is that type-checking terms with redexes is undecidable without it [Dow93]. However, in this thesis we only employ encodings in which the only terms of interest are β-normal, for which type-checking becomes decidable without the extra annotations, rendering them superfluous.

Variables with a type in Kind

In rule ExtCtx allowing to extend a context with a variable $x : T$, we require T to be typed by Type. However, most variants of Dedukti also allow for variables whose type T is in Kind, allowing for instance to type a context $\Gamma_{\mathsf{Nat}} = \mathtt{nat} : \mathsf{Type}, \mathtt{zero} : \mathtt{nat}$. The reason we consider a more restricted typing rule is that variables whose type are in Kind cannot be abstracted and can never leave the context. Because of this, we consider that it makes more sense declaring them as symbols instead of variables, given their global character. So for instance, instead of considering Γ_{Nat}, we can declare the symbols $\mathrm{Nat} : \mathsf{Type}$ and $0 : \mathrm{Nat}$ as being part of the theory $\mathbb{T}$.

Stratification of raw terms

In Figure 2.1, we consider a grammar that specifies one single syntax of terms. However, as we will see in Section 4.1, under some mild hypotheses, the typing judgments of Dedukti impose *a posteriori* a syntactic classification into three syntactic classes, called *objects*, *type families* and *kinds*. In the Edinburgh Logical Framework literature, it is usual to consider a specification of the raw syntax in which this classification is built in [HHP93], so from the start one has three grammars of terms instead of a single one. In the Dedukti literature, this approach is notably taken by Saillard in his PhD thesis [Sai15b].

The main benefits of this alternative approach is that it rules out from the start many terms that would be ill-typed anyway, and it can sometimes allow for reducing the hypotheses of some theorems. However, it also makes the definition of the framework more verbose, and metatheorems such as those of Proposition 2.1 end up being duplicated into three variants, one for each syntactic class. Moreover, most of the Dedukti literature uses a variant with a single grammar, yet another reason for which we chose to stick with this version. However, let us mention that in Chapter 14 we consider a variant of

Dedukti with undirected equations, in which a form of syntactic stratification called *confinement* is used to make possible proving Church-Rosser — see Remark 14.1.

Theories and their typing

Throughout the many previous work on Dedukti, the notion of theory is maybe the one which admits the most possible variations. A first minor choice is whether one separates the symbol declarations from $\mathbb{T}$ into a *(typed) signature* Σ, yielding theories of the form $\mathbb{T} = (\Sigma, \mathcal{R})$ [BDG⁺23], or keep them all together like we do here [Sai15b]. Of course, these two versions are essentially the same.

Then, there are many (non-equivalent) ways one can define what it means for a theory to be well-typed. For instance, Saillard [Sai15b] and Férey [Fer21] ask $\mathbb{T} \triangleright \cdot \vdash T : s$ for all $f : T \in \mathbb{T}$, which has the effect of allowing for symbols whose typing requires themselves[3], a circularity that we prefer to forbid. Throughout his works, Blanqui [BDG⁺23, Bla20] also requires the same condition, but his version of Sym requires one to provide a derivation of $\cdot \vdash T : s$, imposing an implicit (partial) order on symbols that rules out circularities. This is essentially the same as our definition of well-typed theories, except that this order is explicit in our case. However, similarly to Saillard and Férey, we do not integrate in the rule Sym the condition that the symbol's type is well-typed, as we consider this an external verification which should not appear when *using* a theory.

Finally, we have here followed most of the literature in requiring theories to be finite [BDG⁺23, Sai15b, Fer21], however it is a common abuse of the definition to sometimes consider infinite theories.[4] As an alternative to this, Blanqui considers in some of his works a version of Dedukti that allows for infinite theories [Bla20], and because the ordering for typing symbols is implicit in his case, no change to the definition of well-typed theories is required. In our case, we could proceed as Haselwarter and Bauer [HB23], who consider symbols with some well-founded order, and then require each symbol to be well-typed over the theory obtained by considering only smaller symbols. However, recall that Dedukti is developed specifically with the goal of being used in practice and only finite theories can be specified in the implementation, which is why we prefer to not consider infinite theories in this thesis.

Conversion rule with joinability

The conversion rule in our version of Dedukti allows us to deduce $t : U$ from $t : T$ as soon as $T \equiv U$ and U is a well-typed type, a definition that follows most of the literature on Dedukti [BDG⁺23, Sai15b, Fer21, Bla20, CD07, HB21] and on rewriting-based dependent type theories [Bar92, SU06]. However, in some of his works [BGH19, Bla05], Blanqui

[3] For instance, the theory $\mathbb{T} = \mathsf{Tm} : \mathsf{Tm}\ \mathsf{U} \longrightarrow \mathsf{Type},\ \mathsf{U} : \mathsf{Tm}\ \mathsf{U}$ is well-typed in their sense, but the typing of each symbol requires to use rule Sym with both Tm and U.

[4] For instance, Cosineau and Dowek [CD07] define theories as being finite, but then specify an encoding of Pure Type Systems (PTSs) which is infinite as soon as the PTS specification also is.

considers a more restricted version of this rule in which $\equiv$ is replaced with *joinability*, defined as the relation $\longrightarrow^* \circ {}^*\!\longleftarrow$ (writing $\circ$ for the composition of relations). The main benefit of this approach is that it can eliminate the need for confluence in some theorems — see for instance Remark 4.2. However, as already mentioned, using $\equiv$ in the conversion rule is the standard approach in the literature, and it can be counter-intuitive to replace $\equiv$ by a relation that is not necessarily transitive. Nevertheless, most theories used in practice are confluent, in which case $\equiv$ and $\longrightarrow^* \circ {}^*\!\longleftarrow$ become the same.

Rewriting modulo equations

The conversion of theories in Dedukti is specified only by rewrite rules, however sometimes we also require equations which cannot be oriented in a well-behaved manner, such as the commutativity of an operator. To remedy this, some works consider extensions of Dedukti with *rewriting modulo equations*, meaning that conversion is then generated not only by rewrite rules but also by undirected equations [ADJL17, Gen20, Bla22, FBB23]. In Part III we consider one such extension, but which is tailored specifically for *external* undirected equational theories, making it particularly well-suited for encoding Agda-style universe-polymorphism (as show in Chapter 15). Nevertheless, for the first part of this thesis rewriting modulo will not be needed, and so for now we stick with the simpler and more traditional definition of Dedukti.

Part I

Revising the Dedukti Methodology

Chapter 3

Introduction to Part I

Correctness criteria for Dedukti encodings

As previously discussed, the goal of logical frameworks like Dedukti is to allow for the definition of various type theories in a common setting. However, many times these theories one defines are meant to encode other logical systems, usually defined outside of any logical framework. In these situations, what correctness criteria one should employ to ensure that such object-theories are correctly represented by their encodings in the logical framework? In the Dedukti literature, the two main used criteria are *soundness* and *conservativity* [CD07]. When the object-theory is a type theory, these assert that an object-theory typing judgment $t : A$ is derivable for some t if and only if $\mathrm{Tm}\ [\![A]\!]$ is inhabited in the encoding — writing $[\![-]\!]$ for the translation function from the object-theory syntax to the one of the framework.

The problem with conservativity

Whereas soundness, which asserts the direct implication, can usually be easily shown by induction on the object-theory derivation, the same cannot be said for conservativity, which asserts the inverse implication. Indeed, the proof of conservativity has to address a supplementary difficulty, namely that not all framework terms of type $\mathrm{Tm}\ [\![A]\!]$ correspond directly to a valid object-theory term. Thankfully, a correspondence with object-theory terms can in general be found when considering only framework terms that are β-normal. In the case of the Edinburgh Logical Framework (ELF), this allowed for an easy strategy to establish conservativity: because β-reduction is normalizing in ELF, when proving conservativity one can, without loss of generality, suppose that the term witnessing the inhabitation of a type is β-normal.

Unfortunately, conservativity for β-normal forms does not in general imply full conservativity in the case of Dedukti. The reason is that extending the definitional equality with rewrite rules can break the normalization of β, even when the added rules are normalizing by themselves, and also type-preserving and confluent with β. To address this problem, Dowek [Dow17] proposed a notion of model for Dedukti theories which can be used to prove strong normalization of β, and used it to prove normalization for encodings of

Higher-Order Logic and the Calculus of Constructions. Unfortunately, the construction of such models is fraught with technicalities and has to be done in a case-by-case basis for each encoding. This turned out to be a major difficulty for proving conservativity, and most works proposing Dedukti encodings have since left conservativity only as a conjecture [Gen20, HB21, BM21, Fer21, Thi20].[1]

Breaking the normalization of β

We have mentioned that extending the definitional equality with rewrite rules can break the normalization of β, but why is this the case exactly? The reason can be illustrated by looking at how object-theory dependent functions have been traditionally defined in Dedukti, with the rewrite rule

$$\mathrm{Tm}\ (\Pi\ \mathsf{A}\ \mathsf{B}) \longmapsto (x : \mathrm{Tm}\ \mathsf{A}) \to \mathrm{Tm}\ (\mathsf{B}\ x) \tag{$\square$}$$

At first glance, rule ($\square$) can appear reasonable: the symbols and equations one usually declares for defining (strong) dependent functions specify exactly a definitional isomorphism $\mathrm{Tm}\ (\Pi\ A\ B) \simeq (x : \mathrm{Tm}\ A) \to \mathrm{Tm}\ (B\ x)$, and so quotienting it out can be seen simply as a matter of eliminating administrative coercions.[2] However, because this rule "implements" the object-theory β-reduction by the β-reduction of the framework, the crucial point is that the normalization of Dedukti now becomes dependent on that of the encoded object-theory, and in particular can be broken when the latter is non-normalizing. Worse, even when we *know* that the object-theory is normalizing, it is unclear how to deduce from this the normalization of β-reduction in the framework.

This also clarifies why the construction of Dowek's models is bound to be so technical in this case: proving that β normalizes in the encoding becomes at least as hard as proving that β-reduction is normalizing in the encoded system, which is known to be a very difficult task for expressive logical systems.

Our contribution

Our main technical contribution in this part is a new criterion for ensuring that the normalization of β-reduction is preserved when extending Dedukti's definitional equality with rewrite rules. More precisely, our result ensures that this is the case as long as $\mathbb{T}$ is well-typed, $\beta\mathcal{R}_{\mathbb{T}}$ is confluent and the rewrite rules of the theory are not *arrow-producing*, meaning roughly that no function types should appear in right-hand sides. Under these conditions, we will show that the classic proof method [HHP93, GN91, BFG97] of defining a translation to the simply typed λ-calculus can be extended to Dedukti. More explicitly,

[1] With the notable exception of Grienenberger's encoding of ecumenical logic [Gri23], the only work that has since applied Dowek's technique for conservativity.

[2] Similarly to the move from *Coquand-style universes* to *Russell-style universes*, which turns the definitional isomorphism $\mathrm{Tm}\ \mathrm{U} \simeq \mathrm{Ty}$ into a definitional equality $\mathrm{Tm}\ \mathrm{U} \equiv \mathrm{Ty}$ [Ste19].

we will show that the simply typed skeletons of Dedukti's types are preserved by conversion, which will allow us to define a type- and β-reduction-preserving translation to the simply typed λ-calculus, reducing the normalization of β-reduction in Dedukti to its normalization in a system where it is known to hold.

Motivated by our new criterion, we propose a revision of the Dedukti methodology for building encodings and proving conservativity: by adopting a ban of arrow-producing rules in encodings, one can simply apply our criterion to reduce full conservativity to conservativity of β-normal forms, without the need for technical normalization arguments. While arrow-producing rules have been heavily used in previous Dedukti encodings — see for instance the theory proposed by Blanqui *et al.* [BDG⁺23] — they are not really necessary and are even forbidden in logical frameworks such as Uemura's SOGATs [Uem21] and Harper's Equational LF [Har21a].

In order to illustrate how arrow-producing rules can be avoided, we revisit the problem of encoding *Pure Type Systems (PTSs)* in Dedukti. The previous encoding, proposed by Cousineau and Dowek [CD07], employed a rule similar to ($\square$) which prevented the authors from showing full conservativity in their original publication[3]. In Chapter 5, we propose a new encoding of functional Pure Type Systems in Dedukti for which conservativity can be easily shown using the aforementioned strategy. The main difference with Cousineau and Dowek's encoding is that we do not identify $\mathrm{Tm}\ (\Pi\ A\ B)$ and $(x : \mathrm{Tm}\ A) \longrightarrow \mathrm{Tm}\ (B\ x)$, but declare symbols $@$ and λ for going in both directions and a rewrite rule $@\ (\lambda\ \mathtt{t})\ \mathtt{u} \longmapsto \mathtt{t}\ \mathtt{u}$ for representing the object-language β-reduction. It is instructive then to see what happens when encoding a non-normalizing PTS: in this case, $\beta\mathcal{R}_{\mathbb{T}}$ will be non-normalizing, but β by itself will normalize, which is all that is needed to prove conservativity.

Related publication

The content of Part I is adapted from the paper "Adequate and Computational Encodings in the Logical Framework Dedukti" [Fel22a], published in the proceedings of the 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Its name references the fact that the obtained PTS encoding is not only sound and conservative but also *adequate* [HHP93, HL07], meaning that one can exhibit a bijection between PTS terms of type A and Dedukti canonical forms of type $\mathrm{Tm}\ [\![A]\!]$, a property that was not satisfied by Cousineau and Dowek's original encoding. Here we put less emphasis on this because we now realize that this criterion, first proposed in the context of ELF, should be adapted for the case of equational LFs. Indeed, because theories in ELF are pure, the right notion of equality is just the framework's built-in equality, whose equivalence classes are represented by the canonical forms. However, in an equational LF one defines theories whose terms are considered modulo an additional set of equalities,

[3] This was later addressed by Assaf [Ass15], who showed the result using a logical relations technique. As we will see, our methodology allows for a much simpler proof of conservativity without the need for such complex techniques, and as such can be easily adapted to other object-theories.

so adequacy should also take them into account: the right statement would then be a bijection between quotiented PTS terms of type A and quotiented well-typed Dedukti terms of type $\mathrm{Tm}\ [\![A]\!]$. But in this case one can also show Cousineau and Dowek's original encoding of PTSs to be adequate, so we do not consider this to be a selling point for our proposal anymore. We instead prefer to stress the fact that our strategy for proving conservativity is much more straightforward, whereas the traditional approach has proven to be problematic, with many recent work leaving conservativity only as a conjecture [Gen20, HB21, BM21, Fer21, Thi20].

Chapter 4

A Simple Criterion for Strong Normalization of β-reduction

In this chapter we propose a new criterion for establishing the strong normalization of β-reduction in Dedukti. Compared to most normalization criteria for rewriting with dependent types [BGH19, BFG97, Bla05, BvR97], our objective here is to show the normalization of *only* β, even when its union with the other rewrite rules can be potentially non-terminating. This goal is more similar to the the one of Barthe [Bar98], whose criterion can show the normalization of β on extensions of the Calculus of Constructions with term-level rules. However, our criterion also allows for type-level rules, which will be needed in Chapter 5.

As we have already mentioned, our proof works by defining a dependency-erasure map from Dedukti to the simply typed λ-calculus, mapping dependent types to their simply typed skeletons. In order to define the translation, we will therefore need to distinguish the case of Dedukti terms, which are to be mapped to λ-terms, from the case of Dedukti types, which need to be erased into simple types. However, at the level of raw terms, there is not separation between terms and types in Dedukti. The *syntactic stratification theorem* is a standard property of Dedukti allowing to separate them, and is exactly what we need.

We start this chapter by revisiting the syntactic stratification theorem, and give a proof of this result that, unlike other ones versions in the Dedukti literature [Bla01, Fer21], does not depend on subject reduction but instead only in a purely syntactic condition. Then we move to the definition of the translation functions and discuss what conditions we need to impose on $\mathcal{R}_{\mathbb{T}}$ to prove our normalization criterion. Finally, we then conclude this chapter by giving its proof.

4.1 Syntactic stratification

Fix an underlying theory $\mathbb{T}$ for the rest of the section. We call a symbol f *type-level* if its type in $\mathbb{T}$ is of the form $(\vec{x} : \vec{T}) \to \mathsf{Type}$, and define the *syntactic objects* $\mathcal{O}$, *syntactic type* *families* $\mathscr{F}$, and *syntactic kinds* $\mathscr{K}$ by the following grammars. The syntactic classification theorem will then assert that, for every derivable typing judgment $t : T$ in the theory $\mathbb{T}$, there are only three options: we have $t \in \mathscr{K}$ and $T = \mathsf{Kind}$, or $t \in \mathscr{F}$ and $T \in \mathscr{K}$, or $t \in \mathscr{O}$ and $T \in \mathscr{F}$.

$$
\begin{array}{lll}
\boxed{\mathscr{K}} \ni & K ::= \mathsf{Type} \mid (x : F) \to K & \\
\boxed{\mathscr{F}} \ni & F ::= f \mid F\ O \mid x.F \mid (x : F) \to F' & \text{where } f \text{ is type-level} \\
\boxed{\mathscr{O}} \ni & O ::= x \mid f \mid O\ O' \mid x.O & \text{where } f \text{ is not type-level}
\end{array}
$$

The syntactic classes can be easily seen to be closed under substitution for terms in $\mathscr{O}$:

Proposition 4.1 (Closure of $\mathscr{K}$, $\mathscr{F}$ and $\mathscr{O}$ under substitution). *Let $u \in \mathscr{O}$.*

- *If $O \in \mathscr{O}$ then $O[u/x] \in \mathscr{O}$.*
- *If $F \in \mathscr{F}$ then $F[u/x] \in \mathscr{F}$.*
- *If $K \in \mathscr{K}$ then $K[u/x] \in \mathscr{K}$.*

Proof. By easy induction on the definitions of $\mathscr{K}$, $\mathscr{F}$ and $\mathscr{O}$. ■

It would be natural to also try to show closure under reduction, yet this cannot be shown without imposing some restrictions on the rewrite rules[1]. The proofs of the syntactic classification theorem that can be found in the Dedukti literature assume $\beta\mathcal{R}_{\mathbb{T}}$ to satisfy subject reduction and confluence [Fer21, Bla01]. Inspired by a similar proof by Barthe [Bar98], we avoid showing the closure of $\mathscr{F}$ and $\mathscr{K}$ under reduction, and instead define two extensions of these sets for which this property can be shown with weaker hypotheses. In particular, this will enable us to replace subject reduction by a purely syntactic condition in the syntactic classification theorem, which will also allow for our strong-normalization criterion (Theorem 4.2) to rely on weaker hypotheses.

Let us define the auxiliary sets $\hat{\mathscr{K}}$ and $\hat{\mathscr{F}}$ by the following grammar, where T ranges over the full set of Dedukti terms. Note that we have $\mathscr{K} \subset \hat{\mathscr{K}}$ and $\mathscr{F} \subset \hat{\mathscr{F}}$ and $\hat{\mathscr{F}} \cap \hat{\mathscr{K}} = \emptyset$.

$$
\begin{array}{lll}
\boxed{\hat{\mathscr{K}}} \ni & K ::= \mathsf{Type} \mid (x : F) \to K & \\
\boxed{\hat{\mathscr{F}}} \ni & F ::= f \mid F\ T \mid x.F \mid (x : F) \to F' & \text{where } f \text{ is type-level}
\end{array}
$$

We say that a rewrite rule $f\ t_1 \ldots t_k \longmapsto r$ is *type-level* if f is a type-level symbol, and that a type-level rule *weakly preserves type families* if $r \in \hat{\mathscr{F}}$. We then say that $\mathcal{R}$ weakly preserves type families when this is the case for all type-level rules of $\mathcal{R}$. It turns out that this condition is all that is needed to ensure the closure of $\hat{\mathscr{K}}$ and $\hat{\mathscr{F}}$ under rewriting:

[1] For instance, this property does not hold when taking the rule $\mathsf{Nat} \longmapsto \mathsf{Type}$ with $\mathsf{Nat} : \mathsf{Type} \in \mathbb{T}$. Actually, the syntactic classification theorem itself also brakes in this case: we have $x : \mathsf{Nat} \vdash x : \mathsf{Nat}$ and hence $x : \mathsf{Nat} \vdash x : \mathsf{Type}$ by the conversion rule, but the syntactic classification theorem would then imply $\mathsf{Type} \in \mathscr{F}$, which does not hold.

Proposition 4.2 (Closure properties of $\hat{\mathscr{F}}$ and $\hat{\mathscr{K}}$)**.** *Suppose that $\mathcal{R}_{\mathbb{T}}$ weakly preserves type families.*

- *$K \in \hat{\mathscr{K}}$ implies $K[u/x] \in \hat{\mathscr{K}}$, and $F \in \hat{\mathscr{F}}$ implies $F[u/x] \in \hat{\mathscr{F}}$, for all x, u.*
- *$K \in \hat{\mathscr{K}}$ and $K \longrightarrow K'$ implies $K' \in \hat{\mathscr{K}}$, and $F \in \hat{\mathscr{F}}$ and $F \longrightarrow F'$ implies $F' \in \hat{\mathscr{F}}$.*

Proof. The two points follow by induction on the definitions of $\hat{\mathscr{K}}$ and $\hat{\mathscr{F}}$, and for the second point we also do a case analysis on the rewriting position. If the reduction happens at the head, the only possibility is that the applied rewrite rule $f\ t_1 \dots t_k \longmapsto u$ is type-level, and thus $(f\ t_1 \dots t_k)[\theta] \in \hat{\mathscr{F}}$ for some θ. We then have $u \in \hat{\mathscr{F}}$ by hypothesis, and because $\hat{\mathscr{F}}$ is closed under substitution, we conclude $u[\theta] \in \hat{\mathscr{F}}$. ■

Proposition 4.2 then allows us to show the following key lemma, covering the case of the conversion rule in the proof of the syntactic classification theorem.

Lemma 4.1. *Suppose that $\mathcal{R}_{\mathbb{T}}$ weakly preserves type families and $\beta\mathcal{R}_{\mathbb{T}}$ is confluent. If $T \in \mathscr{F} \cup \mathscr{K} \cup \{\mathsf{Kind}\}$ and $T \equiv U$, then $U \in \mathscr{F}$ implies $T \in \mathscr{F}$ and $U \in \mathscr{K}$ implies $T \in \mathscr{K}$.*

Proof. Let us consider the case $U \in \mathscr{F}$, the case $U \in \mathscr{K}$ being symmetric. Then we have $U \in \hat{\mathscr{F}}$, and by confluence we have $T \longrightarrow^* V\ {}^*\!\longleftarrow U$, so by stability of $\hat{\mathscr{F}}$ under reduction we get $V \in \hat{\mathscr{F}}$. Now, if $T \in \mathscr{K}$ this would imply $T \in \hat{\mathscr{K}}$, but by stability of $\hat{\mathscr{K}}$ under reduction we would get $V \in \hat{\mathscr{K}}$, contradiction with the fact that $\hat{\mathscr{K}}$ and $\hat{\mathscr{F}}$ are disjoint. Moreover, if we had $T = \mathsf{Kind}$, then $T \longrightarrow^* V$ would imply $V = \mathsf{Kind}$, contradiction with the fact that $\mathsf{Kind} \notin \hat{\mathscr{F}}$. Therefore, the only remaining possibility is $T \in \mathscr{F}$. ■

We now have all the tools to show the syntactic classification theorem.

Theorem 4.1 (Syntactic classification)**.** *Suppose that $\mathbb{T}$ is well-typed, $\mathcal{R}_{\mathbb{T}}$ weakly preserves type families and $\beta\mathcal{R}_{\mathbb{T}}$ is confluent. If $\Gamma \vdash t : T$ then exactly one of the following holds*

1. *$t \in \mathscr{K}$ and $T = \mathsf{Kind}$*
2. *$t \in \mathscr{F}$ and $T \in \mathscr{K}$*
3. *$t \in \mathscr{O}$ and $T \in \mathscr{F}$*

Proof. Because the sets $\{\mathsf{Kind}\}$ and $\mathscr{K}$ and $\mathscr{F}$ are clearly disjoint, it is clear that at most one of the statements can hold, so it suffices to show that at least one of them holds.

Given two symbols f, f' from the theory, let us write $f \prec f'$ if f occurs before than f' in $\mathbb{T}$. By mapping a type derivation of a judgment to the largest symbol for which rule Sym is applied, we then extend this order to type derivations.

We now show that at least one of the above statements holds, by outer induction on $\prec$[2] and inner structural induction on the derivation. In the following, whenever we say "by

[2] At first sight, this can seem like a poor man's induction on the theory. However, even if $\mathbb{T}$ satisfies confluence it might have prefixes which do not satisfy it, and so it is important that we stay in $\mathbb{T}$ throughout the proof.

induction hypothesis" we mean the inner one, and uses of the outer induction hypothesis will be mentioned explicitly.

- Case Var.

  $$x : A \in \Gamma \; \frac{\Gamma \vdash}{\Gamma \vdash x : A}$$

  From $\Gamma \vdash$ we can extract a strictly smaller derivation of $\Gamma' \vdash A : \mathsf{Type}$ for some Γ', so by i.h. the only possibility is $A \in \mathscr{F}$.

- Case Sym.

  $$f : T \in \mathbb{T} \; \frac{\Gamma \vdash}{\Gamma \vdash f : T}$$

  Because $\mathbb{T}$ is well-typed, we have a derivation of $\cdot \vdash T : s$ in the prefix of $\mathbb{T}$ preceding $f : T$, and so it follows that this derivation is smaller for $\prec$ than the one for $\Gamma \vdash f : T$ we started with. Then, by a form of weakening for theories we obtain a derivation for $\cdot \vdash T : s$ in $\mathbb{T}$ which is still smaller for $\prec$ than the one for $\Gamma \vdash f : T$. Therefore, we can apply the outer induction hypothesis to $\cdot \vdash T : s$. Now, if f is type-level then T is of the form $(\vec{x} : \vec{U}) \longrightarrow \mathsf{Type}$, so the only possibility is $T \in \mathscr{K}$ and $s = \mathsf{Kind}$. Otherwise, if f is not type-level, then T is not of the form $(\vec{x} : \vec{U}) \longrightarrow \mathsf{Type}$, so we must have $T \in \mathscr{F}$ and $s = \mathsf{Type}$.

- Case Conv.

  $$T \equiv U \; \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s}{\Gamma \vdash t : U}$$

  If $s = \mathsf{Type}$, then by the i.h. applied to the second premise we get $U \in \mathscr{F}$, so by applying the i.h. to the first premise, Lemma 4.1 ensures us that the only possible case is $t \in \mathscr{O}$ and $T \in \mathscr{F}$. A similar reasoning proves the case $s = \mathsf{Kind}$.

- Case App.

  $$\frac{\Gamma \vdash t : (x : T) \longrightarrow U \qquad \Gamma \vdash u : T}{\Gamma \vdash t\, u : U[u/x]}$$

  By the i.h. applied to the first premise we have two possibilities: either $t \in \mathscr{O}$ and $(x : T) \longrightarrow U \in \mathscr{F}$, or $t \in \mathscr{F}$ and $(x : T) \longrightarrow U \in \mathscr{K}$. In all cases we have $T \in \mathscr{F}$, so by the i.h. applied to the second premise we get $u \in \mathscr{O}$. Now, if $t \in \mathscr{O}$ and $(x : T) \longrightarrow U \in \mathscr{F}$ then $U \in \mathscr{F}$ and thus $U[u/x] \in \mathscr{F}$ by Proposition 4.1. Similarly, if $t \in \mathscr{F}$ and $(x : T) \longrightarrow U \in \mathscr{K}$ then $U \in \mathscr{K}$ and thus $U[u/x] \in \mathscr{K}$ by Proposition 4.1.

- Case Abs.

$$\frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, x : T \vdash U : s \qquad \Gamma, x : T \vdash t : U}{\Gamma \vdash x.t : (x : T) \to U}$$

By the i.h. applied to the first two premises we get $T \in \mathscr{F}$ and $U \in \mathscr{F} \cup \mathscr{K}$. Therefore, by the i.h. applied to the third premise we have two possibilities: either $t \in \mathscr{O}$ and $U \in \mathscr{F}$, in which case we have $x.t \in \mathscr{O}$ and $(x : T) \to U \in \mathscr{F}$, or $t \in \mathscr{F}$ and $U \in \mathscr{K}$, in which case we have $x.t \in \mathscr{F}$ and $(x : T) \to U \in \mathscr{K}$.

- Case Pi.

$$\frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, x : T \vdash U : s}{\Gamma \vdash (x : T) \to U : s}$$

By the i.h. we have $T \in \mathscr{F}$, and either $U \in \mathscr{F}$ for when $s = \mathsf{Type}$ or $U \in \mathscr{K}$ for when $s = \mathsf{Kind}$. Therefore, if $s = \mathsf{Type}$ we have $(x : T) \to U \in \mathscr{F}$, and if $s = \mathsf{Kind}$ we have $(x : T) \to U \in \mathscr{K}$. ■

Remark 4.1. Note that, by the counterexample of Footnote 1, Theorem 4.1 becomes false when dropping the requirement that $\mathcal{R}_{\mathbb{T}}$ weakly preserves type families. Likewise, confluence cannot be dropped: for instance, in a theory with $f\ \mathsf{t}\ \mathsf{u} \longmapsto \mathsf{t}$ and $f\ \mathsf{t}\ \mathsf{u} \longmapsto \mathsf{u}$ with f not type-level, we have $\mathsf{Type} \equiv \mathsf{Kind}$ and thus $\cdot \vdash \mathsf{Type} : \mathsf{Type}$ by rule Conv. Finally, if the theory is not well-typed, we can have for instance $f : \mathsf{Kind} \to \mathsf{Kind} \in \mathbb{T}$ and thus $\cdot \vdash f : \mathsf{Kind} \to \mathsf{Kind}$ by rule Sym, also breaking the theorem. □

4.2 The translation functions

Before defining the translation functions used in our proof of normalization, let us recall the syntax of the simply typed λ-calculus and of the simple types on a single base type:

$$\boxed{\mathsf{Ty}_\lambda} \ni \qquad \sigma ::= * \mid \sigma \to \sigma'$$

$$\boxed{\mathsf{Tm}_\lambda} \ni \qquad t, u ::= x \mid \lambda x.t \mid t\ u$$

Let us also define γ_{dk} as the simply typed context containing the declaration type : $*$ and, for each simple type σ, the declaration $\pi_\sigma : * \to (\sigma \to *) \to *$. We can now define the term-translation function $|-|$ and the dependency-erasure function $\|-\|$ by the clauses of Figure 4.1.[3][4] The syntactic stratification theorem will be essential later to show that

[3] Note that here we define $\|-\|$ over $\hat{\mathscr{F}} \cup \hat{\mathscr{K}} \cup \{\mathsf{Kind}\}$ instead of $\mathscr{F} \cup \mathscr{K} \cup \{\mathsf{Kind}\}$, which would have been the most natural option. This is because we will need its domain to be stable under reduction, which in the case of $\hat{\mathscr{F}} \cup \hat{\mathscr{K}} \cup \{\mathsf{Kind}\}$ is ensured by Proposition 4.2.

[4] Had we chosen to use a variant of Dedukti with domain-annotated abstractions, we could apply Harper *et al.*'s [HHP93] trick of defining $|x^T.t|$ as $(\lambda zx.|t|)|T|$ for some z fresh — see the previous version of our work where this is done [Fel22a].

$$\begin{array}{ll}
\|{-}\| : \hat{\mathscr{F}} \cup \hat{\mathscr{K}} \cup \{\mathsf{Kind}\} \longrightarrow \mathsf{Ty}_\lambda & \|f\| := * \\
\|\mathsf{Kind}\| := * & \|F\ T\| := \|F\| \\
\|\mathsf{Type}\| := * & \|x.F\| := \|F\| \\
\|(x : F) \longrightarrow K\| := \|F\| \longrightarrow \|K\| & \|(x : F) \longrightarrow F'\| := \|F\| \longrightarrow \|F'\|
\end{array}$$

$$\begin{array}{ll}
|-| : \mathscr{O} \cup \mathscr{F} \cup \mathscr{K} \longrightarrow \mathsf{Tm}_\lambda & |x.F| := \lambda x.|F| \\
|\mathsf{Type}| := \mathsf{type} & |(x : F) \longrightarrow F'| := \pi_{\|F\|}\ |F|\ (\lambda x.|F'|) \\
|(x : F) \longrightarrow K| := \pi_{\|F\|}\ |F|\ (\lambda x.|K|) & |x| := x \\
|f| := f & |O\ O'| := |O||O'| \\
|F\ O| := |F||O| & |x.O| := \lambda x.|O|
\end{array}$$

Figure 4.1: Term-translation and dependency-erasure functions

these functions are defined for all well-typed terms and types. We extend $\|{-}\|$ naturally to contexts and theories whose types are all in $\hat{\mathscr{F}} \cup \hat{\mathscr{K}} \cup \{\mathsf{Kind}\}$, by simply ignoring rewrite rule declarations.

In order to show the normalization of β-reduction in Dedukti, we will first show that the translation preserves typing and that $|-|$ preserves β-reduction sequences, which will then allow us to derive our result by appealing to the strong-normalization of the simply typed λ-calculus. The main difficulty is dealing with the conversion rule when showing preservation of typing, which requires proving that convertible types are mapped by $\|{-}\|$ into the same simple type. Whereas this can be easily seen to hold when β is the only rewrite rule, it is not very hard to find counterexamples when other rules are allowed.

Example 4.1. Let Tm be a type-level symbol, and consider the rule

$$\mathrm{Tm}\ (\Pi\ \mathsf{A}\ \mathsf{B}) \longmapsto (x : \mathrm{Tm}\ \mathsf{A}) \longrightarrow \mathrm{Tm}\ (\mathsf{B}\ x)$$

traditionally used to encode dependent functions in Dedukti [CD07]. We then have

$$\mathrm{Tm}\ (\Pi\ \mathrm{Nat}\ (x.\mathrm{Nat})) \equiv \mathrm{Tm}\ \mathrm{Nat} \longrightarrow \mathrm{Tm}\ \mathrm{Nat}$$

but $\|\mathrm{Tm}\ (\Pi\ \mathrm{Nat}\ (x.\mathrm{Nat}))\| = *$ and $\|\mathrm{Tm}\ \mathrm{Nat} \longrightarrow \mathrm{Tm}\ \mathrm{Nat}\| = * \longrightarrow *$. □

The previous example motivates the following definition, which will allow us to avoid situations of this kind. A type-level rule is said to be *not arrow-producing* if its right-hand side is in the following grammar, where T ranges over all Dedukti terms.

$$\boxed{\widetilde{\mathscr{F}}} \ni \qquad F ::= f \mid F\ T \mid x.F \qquad \text{where } f \text{ is type-level}$$

Note that this grammar is almost the same as the one defining $\hat{\mathscr{F}}$, with the difference that we forbid the use of arrow types. We then say that $\mathcal{R}_{\mathbb{T}}$ is not arrow-producing when this is the case for all of its type-level rules — which implies in particular that $\mathcal{R}_{\mathbb{T}}$ weakly preserves type families. As we will see, this condition, together with confluence of $\beta\mathcal{R}_{\mathbb{T}}$ and well-typedness of $\mathbb{T}$, will be sufficient to establish the strong normalization of β-reduction.

4.3 Proving the criterion

In order to show that the translation preserves typing, we will need the following lemma, whose proof strongly relies on the assumption that $\mathcal{R}_{\mathbb{T}}$ is not arrow-producing.

Lemma 4.2 (Invariance of $\|-\|$). *Suppose that $\mathcal{R}_{\mathbb{T}}$ is not arrow-producing.*

1. *If $T \in \hat{\mathscr{F}} \cup \hat{\mathscr{K}} \cup \{\mathsf{Kind}\}$ then $\|T\| = \|T[t/x]\|$ for all t.*

2. *If $T \in \hat{\mathscr{F}} \cup \hat{\mathscr{K}} \cup \{\mathsf{Kind}\}$ and $T \longrightarrow T'$ then $\|T\| = \|T'\|$*

3. *If $\beta\mathcal{R}_{\mathbb{T}}$ is confluent and $T, U \in \hat{\mathscr{F}} \cup \hat{\mathscr{K}} \cup \{\mathsf{Kind}\}$ and $T \equiv U$ then $\|T\| = \|U\|$.*

Proof. First note that Proposition 4.2 ensures us, in the first two points, that $T[t/x]$ and T' are indeed in the domain of $\|-\|$.

1. By induction on the definition of $\|-\|$.

2. By induction on the definition of $\|-\|$ and case analysis on the rewrite position. The only interesting case is when the reduction happens at the head. There are then two possibilities. If $T = (x.T_1)\ T_2 \longrightarrow_{\beta} T_1[T_2/x]$ we conclude $\|T\| = \|T_1\| = \|T_1[T_2/x]\|$ using the first point. Otherwise we have $T = (f\ t_1 \dots t_k)[\theta] \longrightarrow r[\theta]$ for some type-level rule $f\ t_1 \dots t_k \longmapsto r$, in which case we have $\|T\| = *$. Then, because $\mathcal{R}_{\mathbb{T}}$ is not arrow producing, we can show that $\|r\| = *$ by induction on the definition of $\widetilde{\mathscr{F}}$, which then implies $\|r[\theta]\| = *$ by the first point.

3. By confluence we have $T \longrightarrow^* V\ {}^*\!\!\longleftarrow U$, so we conclude by iterating point 2. ■

Let us write $\gamma \vdash_{\lambda} t : \sigma$ for the typing judgment of the simply typed λ-calculus.

Proposition 4.3 (Translation preserves typing). *Suppose that $\mathbb{T}$ is well-typed, $\mathcal{R}_{\mathbb{T}}$ is not arrow-producing and $\beta\mathcal{R}_{\mathbb{T}}$ is confluent. Then $\Gamma \vdash t : T$ implies $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_{\lambda} |t| : \|T\|$.*

Proof. First note that, by Theorem 4.1, whenever we have $\Gamma \vdash t : T$ then $|t|$ and $\|T\|$ are defined. Moreover, because $\Gamma \vdash t : T$ implies $\Gamma \vdash$, which implies $\Gamma \vdash U : \mathsf{Type}$ for all $x : U \in \Gamma$, then Theorem 4.1 also ensures that $\|\Gamma\|$ is defined. Finally, because we suppose that the theory is well-typed, for all $f : U \in \mathbb{T}$ we have $\cdot \vdash U : s$ in some prefix of $\mathbb{T}$. But then by a form of weakening for theories, we obtain $\cdot \vdash U : s$ in $\mathbb{T}$, so by Theorem 4.1 we conclude that $\|U\|$ is indeed well-defined.

We now show the result by induction on the derivation of $\Gamma \vdash t : T$.

- Case Type.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Type} : \mathsf{Kind}}$$

We have $(\mathsf{type} : *) \in \gamma_{\mathsf{dk}}$, so we conclude $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash \mathsf{type} : *$.

- Case Var.

$$x : T \in \Gamma \, \frac{\Gamma \vdash}{\Gamma \vdash x : T}$$

We have $x : \|T\| \in \|\Gamma\|$, so we conclude $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda x : \|T\|$.

- Case Sym.

$$f : T \in \mathbb{T} \, \frac{\Gamma \vdash}{\Gamma \vdash f : T}$$

We have $f : \|T\| \in \|\mathbb{T}\|$, so we conclude $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda f : \|T\|$.

- Case Conv.

$$T \equiv U \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s}{\Gamma \vdash t : U}$$

By i.h. we have $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda |t| : \|T\|$, and by Theorem 4.1 we also have $U \in \mathscr{F} \cup \mathscr{K}$, so we conclude by applying Lemma 4.2 to get $\|T\| = \|U\|$.

- Case App.

$$\frac{\Gamma \vdash t : (x : T) \to U \qquad \Gamma \vdash u : T}{\Gamma \vdash t\ u : U[u/x]}$$

By i.h. we have $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda |t| : \|T\| \to \|U\|$ and $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda |u| : \|U\|$, so we obtain $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda |t|\ |u| : \|U\|$, and we conclude by applying Lemma 4.2 to get $\|U\| = \|U[u/x]\|$.

- Case Abs.

$$\frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, x : T \vdash U : s \qquad \Gamma, x : T \vdash t : U}{\Gamma \vdash x.t : (x : T) \to U}$$

By i.h. we have $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\|, x : \|T\| \vdash_\lambda |t| : \|U\|$, so we conclude $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda \lambda x.|t| : \|T\| \to \|U\|$.

- Case Pi.

$$\frac{\Gamma \vdash T : \mathsf{Type} \qquad \Gamma, x : T \vdash U : s}{\Gamma \vdash (x : T) \to U : s}$$

By i.h. we get $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda |T| : * $ and $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\|, x : \|T\| \vdash_\lambda |U| : *$, so we can show $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda \lambda x.|U| : \|T\| \to *$ and conclude $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda \pi_{\|T\|}\ |T|\ (\lambda x.|U|) : *$. ■

We now show the preservation of β-reduction by $| - |$.

Proposition 4.4 ($| - |$ preserves β-reduction). *Suppose that $\mathcal{R}_\mathbb{T}$ is not arrow-producing. If $t \longrightarrow_\beta t'$ and $t \in \mathscr{K} \cup \mathscr{F} \cup \mathscr{O}$ then $t' \in \mathscr{K} \cup \mathscr{F} \cup \mathscr{O}$ and $|t| \longrightarrow_\beta |t'|$.*

Proof. By induction on the definition of $| - |$ and case analysis on the position of the reduction. The most interesting case is a reduction at the head, for which we then need to prove that $u \in \mathscr{F} \cup \mathscr{O}$ and $v \in \mathscr{O}$ imply $|u|[|v|/x] = |u[v/x]|$, by induction on u and using Lemma 4.2. The other cases mostly follow from the i.h., but for the case $(x : T) \to U \longrightarrow (x : T') \to U$ with $T \in \mathscr{F}$ and $U \in \mathscr{F} \cup \mathscr{K}$ we also need Lemma 4.2 to ensure $\pi_{\|T\|} = \pi_{\|T'\|}$. ■

Our criterion now follows as a simple corollary of Propositions 4.3 and 4.4 and the strong normalization of β-reduction in the simply typed λ-calculus [SU06, Theorem 3.5.5].

Theorem 4.2 (Strong-normalization of β-reduction). *If $\mathbb{T}$ is well-typed, $\mathcal{R}_\mathbb{T}$ is not arrow-producing and $\beta\mathcal{R}_\mathbb{T}$ is confluent, then β is strongly normalizing for the well-typed terms of $\mathbb{T}$.*

Proof. Let t be a term for which we have $\Gamma \vdash t : T$ and consider a reduction sequence $t \longrightarrow_\beta t_1 \longrightarrow_\beta t_2 \longrightarrow_\beta \dots$ starting at t. By Proposition 4.3 we have $\gamma_{\mathsf{dk}}, \|\mathbb{T}\|, \|\Gamma\| \vdash_\lambda |t| : \|T\|$, so $|t|$ is strongly normalizing for β-reduction. But by Proposition 4.4 we get $|t| \longrightarrow_\beta |t_1| \longrightarrow_\beta |t_2| \longrightarrow_\beta \dots$, implying that this sequence is finite. ■

Remark 4.2. In Remark 4.1 we saw that the hypothesis of confluence cannot be dropped in Theorem 4.1. By a similar argument, we can show the same for the above result: by taking the rules $f\ \mathsf{t}\ \mathsf{u} \longmapsto \mathsf{t}$ and $f\ \mathsf{t}\ \mathsf{u} \longmapsto \mathsf{u}$ for some f not type-level, and assuming $\mathsf{Nat} : \mathsf{Type} \in \mathbb{T}$, we can show $\mathsf{Nat} \equiv \mathsf{Nat} \to \mathsf{Nat}$, allowing us to encoded untyped λ-terms as Dedukti terms of type Nat. However, let us note that, by moving to a variant of Dedukti in which $\equiv$ is replaced by $\longrightarrow^* \circ {}^*\!\longleftarrow$ in rule Conv, the hypothesis of confluence could be dropped in both Theorems 4.1 and 4.2[5]. □

[5] The author thanks Frédéric Blanqui for pointing this out.

Chapter 5

Encoding Pure Type Systems in Dedukti

Among the many type formers one can find in type theories, two appear very recurrently: dependent function types and universes. The class of type theories only featuring these type formers was first identified by Berardi and Terlouw, and is nowadays known as *Pure Type Systems* (or PTSs) [Bar91, Bar92]. Since their introduction, Pure Type Systems have been thoroughly studied by many authors and have become a central object of study in the type theory community. They have in particular provided a uniform framework for studying various universe hierarchies, and isolating previously known universe paradoxes from unimportant syntactic details [Coq86].

In this chapter, we illustrate our new methodology for Dedukti encodings with this important class of type theories. We start by defining the variant of PTSs we consider before moving to the definition of the encoding. This is followed by the proofs of soundness and conservativity, the last one being the main contribution of this chapter.

5.1 Pure Type Systems

The definition of Pure Type Systems admits many variations in the literature: with typed or untyped equality [Ada06, SH12], with more or less annotations [MW96, BS00], with $\beta\eta$-equality or just β-equality [Geu92], with our without explicit contexts [GKMW10], etc. In this work, we mostly stick to the more well-established definition of PTSs [BDS13], except that we employ a *fully annotated syntax*. If we recall that the dependent function type $\Pi x : A.B$ is a *parametrized* type, then we can notice that the usual notations for abstraction $\lambda x : A.t$ and application $t\ u$ omit either some or all of the parameters A and B. Moreover, because in PTSs the function type can be used across multiple universe levels, in order to be fully explicit with the parameters, we would also need to annotate Π, abstraction and application with levels. Therefore, our fully annotated version of PTSs will make all of this information explicit. More precisely, given a set $\mathscr{L}$ of *universe levels*[1],

[1] Usually called *sorts* in the PTS literature.

$\boxed{\Gamma \vdash_{\mathsf{PTS}}}$

$$\textsc{EmptyCtx}\ \frac{}{\cdot \vdash_{\mathsf{PTS}}} \qquad \textsc{ExtCtx}\ \ l \in \mathscr{L}\ \frac{\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l}{\Gamma, x : A \vdash_{\mathsf{PTS}}}$$

$\boxed{\Gamma \vdash_{\mathsf{PTS}} t : A}$

$$\textsc{Var}\ \ x : A \in \Gamma\ \frac{\Gamma \vdash_{\mathsf{PTS}}}{\Gamma \vdash_{\mathsf{PTS}} x : A} \qquad \textsc{Pi}\ \ (l, l', l'') \in \mathscr{R}\ \frac{\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l \qquad \Gamma, x : A \vdash_{\mathsf{PTS}} B : \mathsf{U}_{l'}}{\Gamma \vdash_{\mathsf{PTS}} \Pi_{l,l'} x : A.B : \mathsf{U}_{l''}}$$

$$\textsc{Univ}\ \ (l, l') \in \mathscr{A}\ \frac{\Gamma \vdash_{\mathsf{PTS}}}{\Gamma \vdash_{\mathsf{PTS}} \mathsf{U}_l : \mathsf{U}_{l'}} \qquad \textsc{Abs}\ \ (l, l', l'') \in \mathscr{R}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l \qquad \Gamma, x : A \vdash_{\mathsf{PTS}} B : \mathsf{U}_{l'} \\ \Gamma, x : A \vdash_{\mathsf{PTS}} t : B\end{array}}{\Gamma \vdash_{\mathsf{PTS}} \lambda^{x:A.B}_{l,l'} x.t : \Pi_{l,l'} x : A.B}$$

$$\textsc{Conv}\ \ \begin{array}{c} l \in \mathscr{L} \\ A \equiv_\beta B \end{array}\ \frac{\Gamma \vdash_{\mathsf{PTS}} t : A \qquad \Gamma \vdash_{\mathsf{PTS}} B : \mathsf{U}_l}{\Gamma \vdash_{\mathsf{PTS}} t : B} \qquad \textsc{App}\ \ (l, l', l'') \in \mathscr{R}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l \qquad \Gamma, x : A \vdash_{\mathsf{PTS}} B : \mathsf{U}_{l'} \\ \Gamma \vdash_{\mathsf{PTS}} t : \Pi_{l,l'} x : A.B \qquad \Gamma \vdash_{\mathsf{PTS}} u : A\end{array}}{\Gamma \vdash_{\mathsf{PTS}} t @^{x:A.B}_{l,l'} u : B[u/x]}$$

Figure 5.1: Typing rules for fully annotated Pure Type Systems

we consider the following syntax of raw terms and contexts, where l and l' range over $\mathscr{L}$.

$$\begin{array}{lrl} \boxed{\mathsf{Tm}_{\mathsf{PTS}}} \ni & t, u, A, B ::= & x \mid \mathsf{U}_l \mid \Pi_{l,l'} x : A.B \mid \lambda^{x:A.B}_{l,l'} x.t \mid t @^{x:A.B}_{l,l'} u \\ \boxed{\mathsf{Ctx}_{\mathsf{PTS}}} \ni & \Gamma, \Delta ::= & \cdot \mid \Gamma, x : A \end{array}$$

In this setting, β-reduction is defined as the closure under context of the following reduction rule. Note that we consider a *linearized* variant of the expected non-left-linear rule, which would be non-confluent on untyped terms, whereas the following rule is confluent by orthogonality [MN98].[2]

$$(\lambda^{x:A'.B'}_{l,l'} x.t) @^{x:A.B}_{l,l'} u \longrightarrow_\beta t[u/x]$$

Given two relations $\mathscr{A} \subseteq \mathscr{L}^2$ and $\mathscr{R} \subseteq \mathscr{L}^3$, we define typing by the rules in Figure 5.1. The triple $\mathscr{S} = (\mathscr{L}, \mathscr{A}, \mathscr{R})$ is called a *PTS specification*, which is said to be *functional* when

[2] The reader might complain that the rule is still non-left-linear, as l and l' occur twice. Because universe levels do not partake in rewriting, the annotations l, l' need not be seen as arguments but instead can be seen as part of the names of abstraction and application, which thus become (possibly infinite) $\mathscr{L}^2$-indexed families of symbols, making β-reduction a $\mathscr{L}^2$-indexed family of left-linear rules.

$\mathscr{A}$ and $\mathscr{R}$ are functional relations, when seeing $\mathscr{R}$ as $\mathscr{R} \subseteq \mathscr{L}^2 \times \mathscr{L}$. When defining our encoding, we will only consider functional specifications, given that non-functional specifications are generally of lesser interest and not much used in practice.

Our use of a fully annotated syntax will be necessary to define the translation function of the encoding.[3] We have already mentioned that other authors have already considered some variants of PTSs with different amounts of annotations (such as Siles and Herbelin [SH12], Mellies and Werner [MW96], and Barthe and Sørensen [BS00]), yet none of these variants correspond exactly to the one we employ here. Therefore, we revisited the basic metatheory of PTSs for this variant in a technical report [Fel22b], and have found that all the usual PTSs metaproperties are preserved:

Proposition 5.1 (Basic PTS properties)**.** *The following hold for all PTS specifications $\mathcal{S}$.*[4]

Confluence *Given $t, u \in \mathsf{Tm}_{\mathsf{PTS}}$, if $t \equiv_\beta u$ then $t \longrightarrow^*_\beta v \; {}^*_\beta\!\!\longleftarrow u$ for some $v \in \mathsf{Tm}_{\mathsf{PTS}}$.*

Weakening *Suppose $\Gamma \sqsubseteq \Gamma'$ and $\Gamma' \vdash_{\mathsf{PTS}}$. Then $\Gamma \vdash_{\mathsf{PTS}} t : A$ implies $\Gamma' \vdash_{\mathsf{PTS}} t : A$.*

Substitution property *If $\Gamma, x : B, \Gamma' \vdash_{\mathsf{PTS}} t : A$ and $\Gamma \vdash_{\mathsf{PTS}} u : B$ then $\Gamma, \Gamma'[u/x] \vdash_{\mathsf{PTS}} t[u/x] : A[u/x]$.*

Validity *If $\Gamma \vdash_{\mathsf{PTS}} t : A$ then either $A = \mathsf{U}_l$ or $\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l$ for some $l \in \mathscr{L}$.*

Uniqueness of types *If $\mathcal{S}$ is functional, then $\Gamma \vdash_{\mathsf{PTS}} t : A$ and $\Gamma \vdash_{\mathsf{PTS}} t : B$ imply $A \equiv B$.*

Subject reduction *If $\Gamma \vdash_{\mathsf{PTS}} t : A$ and $t \longrightarrow_\beta t'$ then $\Gamma \vdash_{\mathsf{PTS}} t' : A$.*

Extended conversion *Let us write $\Gamma \vdash_{\mathsf{PTS}} B\ \mathsf{type}$ when $\Gamma \vdash_{\mathsf{PTS}} B : \mathsf{U}_l$ or $B = \mathsf{U}_l$ for some $l \in \mathscr{L}$. Then $\Gamma \vdash_{\mathsf{PTS}} t : A$ and $\Gamma \vdash_{\mathsf{PTS}} B\ \mathsf{type}$ with $A \equiv_\beta B$ imply $\Gamma \vdash_{\mathsf{PTS}} t : B$.*

We have also established the following equivalence, showing that the move to a fully annotated syntax does not morally change the type system when the specification is functional. Once again, we refer to the technical report for the proof [Fel22b].

Theorem 5.1 (Equivalence between fully annotated PTSs and regular PTSs)**.** *Let us write $|-|$ for the erasure function from the fully annotated syntax to the usual PTS syntax. For all functional specifications, the judgment $\Gamma \vdash_{\mathsf{PTS}} t : A$ holds for the regular PTS definition [Bar92] iff we have $\Gamma' \vdash_{\mathsf{PTS}} t' : A'$ for some Γ', t', A' with $|\Gamma'| = \Gamma$ and $|t'| = t$ and $|A'| = A$.*

[3] Switching to a more annotated syntax is also needed in some previous Dedukti encodings [HB21].

[4] Extended conversion is actually not proven in the technical report [Fel22b], however it is an easy consequence of validity, confluence, subject reduction, and rule Conv.

5.2 The encoding

In this section we define our encoding of functional Pure Type Systems. Therefore, from now on all considered PTS specifications are functional.

We proceed in two steps: first we specify the underlying theory of the encoding, and then we define the translation function from the PTS syntax to the one of the framework.

Judgment forms and universes

To define the theory of our encoding, first recall that the judgments-as-types principle specifies that object-logic judgment forms are to be represented in the framework by type-level symbols[5], so that the subject of a judgment becomes then an inhabitant of its representing type. In the case of PTSs, the only judgment form is $\square : A$, where we have replaced the subject by $\square$, which should thus be represented as $\mathrm{Tm}\ [\![A]\!]$ (writing $[\![A]\!]$ for the representation of A) so that PTS terms for which $t : A$ is derivable correspond to inhabitants of $\mathrm{Tm}\ [\![A]\!]$. But what should then be the type of the symbol Tm? As a simplifying step, let us first assume $\mathscr{L}$ to be a singleton and $\mathscr{A} = \mathscr{L}^2$. Then the type A in $t : A$ is itself a term satisfying $A : \mathsf{U}$, and so its translation should inhabit $\mathrm{Tm}\ [\![\mathsf{U}]\!]$. If we define the symbol U as the translation of U, then the above reasoning suggest that Tm should have type $\mathrm{Tm}\ \mathrm{U} \to \mathrm{Type}$, meaning that the type of Tm refers to Tm itself!

The aforementioned circularity is due to the fact that the usual syntactic presentations of PTSs use *Russell-style universes*, in which a term of type U is automatically a type. This strict identification has led some think that Russell-style universes cannot be expressed in logical frameworks, and that the only way out is switching to *Tarski-style universes*. This is in fact not the case, and the strategy to remove this circularity actually has already been known for a while [Ste19][6]: we start by introducing a new symbol $\mathrm{Ty} : \mathrm{Type}$ as a placeholder for $\mathrm{Tm}\ \mathrm{U}$, then we declare the symbols $\mathrm{U} : \mathrm{Ty}$ and $\mathrm{Tm} : \mathrm{Ty} \to \mathrm{Type}$, and we conclude by closing the loop with a rewrite rule $\mathrm{Tm}\ \mathrm{U} \longmapsto \mathrm{Ty}$. A posteriori we then indeed have $\mathrm{Tm} : \mathrm{Tm}\ \mathrm{U} \to \mathrm{Type}$, but crucially not by definition.

We now need to get rid of the assumption that $\mathscr{L}$ is a singleton, which can easily be done by replacing Ty and Tm by the symbols $\mathrm{Ty}_l : \mathrm{Type}$ and $\mathrm{Tm}_l : \mathrm{Ty}_l \to \mathrm{Type}$ for $l \in \mathscr{L}$, and replacing U and $\mathrm{Tm}\ \mathrm{U} \longmapsto \mathrm{Ty}$ by the symbols $\mathrm{U}_l : \mathrm{Ty}_{l'}$ and rules $\mathrm{Tm}_{l'}\ \mathrm{U}_l \longmapsto \mathrm{Ty}_l$ for $(l, l') \in \mathscr{A}$. The judgment $\square : A$ can then be represented by the type $\mathrm{Tm}_l\ [\![A]\!]$ for l the universe level of A. This is almost correct, however we should take care to notice that A might not have a level if $A = \mathsf{U}_l$ with l a *maximal level* (meaning that there is no l' with $(l, l') \in \mathscr{A}$). Fortunately, we can easily solve this by adding a top layer $\mathrm{Ty}_\top : \mathrm{Type}$ and $\mathrm{Tm}_\top : \mathrm{Ty}_\top \to \mathrm{Type}$, and adding $\mathrm{U}_l : \mathrm{Ty}_\top$ and $\mathrm{Tm}_\top\ \mathrm{U}_l \longmapsto \mathrm{Ty}_l$ for all maximal levels l. Now, if l is a maximal level, the PTS judgment $\square : \mathsf{U}_l$ can be represented in the framework

[5] Except for any context or equality judgments, which are handled directly by the logical framework.

[6] And the method has recently been generalized by Altenkirch *et al.* [AKvV23], who have dubbed it "The Münchhausen Method".

by the type $\mathrm{Tm}_\top\ \mathrm{U}_l$.

If we define $\mathscr{L}_\top := \mathscr{L} \cup \{\top\}$, and l^+ as either $\top$ when l is a maximal level or l' when $(l, l') \in \mathscr{A}$ — which is unique because we assume the specification to be functional — then the above declarations can be resumed as:

$$
\begin{array}{ll}
\mathrm{Ty}_{l_\top} : \mathsf{Type} & \text{for } l_\top \in \mathscr{L}_\top \\
\mathrm{Tm}_{l_\top} : \mathrm{Ty}_{l_\top} \to \mathsf{Type} & \text{for } l_\top \in \mathscr{L}_\top \\
\mathrm{U}_l : \mathrm{Ty}_{l^+} & \text{for } l \in \mathscr{L} \\
\mathrm{Tm}_{l^+}\ \mathrm{U}_l \longmapsto \mathrm{Ty}_l & \text{for } l \in \mathscr{L}
\end{array}
$$

Remark 5.1. If we were to apply the judgments-as-types principle backwards to the above declarations, we would get a type system with two typing judgments $\Gamma \vdash_{\mathsf{PTS}} t :_{l_\top} A$ and $\Gamma \vdash_{\mathsf{PTS}} A\ \mathrm{type}_{l_\top}$ for $l_\top \in \mathscr{L}_\top$. If we consider these as the ground truth, we realize that the usual definition of PTSs introduces two simplifications: first, it omits the annotation $l_\top$ in the term typing judgment by writing $\Gamma \vdash_{\mathsf{PTS}} t : A$ instead of $\Gamma \vdash_{\mathsf{PTS}} t :_{l_\top} A$, and it uses the fact that the judgments $\Gamma \vdash_{\mathsf{PTS}} A :_{l^+} \mathsf{U}_l$ and $\Gamma \vdash_{\mathsf{PTS}} A\ \mathrm{type}_l$ classify the same elements (as specified by the rewrite rule $\mathrm{Tm}_{l^+}\ \mathrm{U}_l \longmapsto \mathrm{Ty}_l$) to replace all the occurrences of the former by the latter. Note however that this last simplification cannot be performed for judgments $\Gamma \vdash_{\mathsf{PTS}} A\ \mathrm{type}_\top$, yet the usual definition of PTSs still scraps completely the judgments $\Gamma \vdash_{\mathsf{PTS}} A\ \mathrm{type}_{l_\top}$ even for $l_\top = \top$. The result of this is a well-known oddity: if l is a maximal level, then there can be terms A for which $\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l$ is derivable, yet no PTS judgment allows one to establish the well-formedness of U_l itself.[7] The encoding of PTSs in a logical framework thus allows us to identify the cause of this problem as an oversimplification in the usual definition of PTSs. □

Dependent functions

Up until this point we have encoded only universes in the theory, in a way that mostly follows the original PTS encoding in Dedukti by Cousineau and Dowek [CD07]. We now continue by defining dependent functions, and start by declaring a family of symbols

$$\Pi_{l,l'} : (\mathsf{A} : \mathrm{Ty}_l) \to (\mathsf{B} : \mathrm{Tm}_l\ \mathsf{A} \to \mathrm{Ty}_{l'}) \to \mathrm{Ty}_{l''} \qquad \text{for } (l, l', l'') \in \mathscr{R}$$

to represent the type former Π. This is now the point in which we depart from the usual Dedukti strategy introduced by Cousineau and Dowek: they would continue here by adding the rewrite rules $\mathrm{Tm}_{l''}\ (\Pi_{l,l'}\ \mathsf{A}\ \mathsf{B}) \longmapsto (x : \mathrm{Tm}_l\ \mathsf{A}) \to \mathrm{Tm}_{l'}\ (\mathsf{B}\ x)$ for $(l, l', l'') \in \mathscr{R}$, identifying framework terms of type of $\mathrm{Tm}_{l''}\ (\Pi_{l,l'}\ A\ B)$ with framework terms of type $(x : \mathrm{Tm}_l\ A) \to \mathrm{Tm}_{l'}\ (B\ x)$.

This identification can at first appear reasonable: the typing rules for application and abstraction, along with the β-rule, establish exactly a retraction of terms t satisfying

[7] Note that Dedukti itself, which can be seen as a *PTS modulo* [Bla01], also suffers from this oddity.

$\Gamma \vdash_{\mathsf{PTS}} t : \Pi_{l,l'} x : A.B$ onto terms t satisfying $\Gamma, x : A \vdash_{\mathsf{PTS}} t : B$, which moreover becomes an isomorphism when taking the η-rule. However, as discussed in Chapter 3, even if this rewrite rule is semantically sensible, it can also break the normalization of β-reduction in the framework, making the proof of conservativity of the encoding very hard. Instead, our goal is to only use rewrite rules that are not arrow-producing, which will then allow us to use Theorem 4.2 to ensure that β-reduction is still normalizing in the framework.

Our way to continue here will thus be the least creative one possible: we simply add families of symbols $\lambda_{l,l'}$ and $@_{l,l'}$ for representing the PTS abstraction and application, and a family of rewrite rules for representing the PTS β-reduction, yielding the following declarations:

$$\begin{aligned}
&\lambda_{l,l'} : (\mathsf{A} : \mathrm{Ty}_l) \longrightarrow (\mathsf{B} : \mathrm{Tm}_l\ \mathsf{A} \longrightarrow \mathrm{Ty}_{l'}) \longrightarrow \\
&\quad ((x : \mathrm{Tm}_l\ \mathsf{A}) \longrightarrow \mathrm{Tm}_{l'}\ (\mathsf{B}\ x)) \longrightarrow \mathrm{Tm}_{l''}\ (\Pi_{l,l'}\ \mathsf{A}\ \mathsf{B}) && \text{for } (l, l', l'') \in \mathscr{R} \\
&@_{l,l'} : (\mathsf{A} : \mathrm{Ty}_l) \longrightarrow (\mathsf{B} : \mathrm{Tm}_l\ \mathsf{A} \longrightarrow \mathrm{Ty}_{l'}) \longrightarrow \\
&\quad (\mathsf{t} : \mathrm{Tm}_{l''}\ (\Pi_{l,l'}\ \mathsf{A}\ \mathsf{B})) \longrightarrow (\mathsf{u} : \mathrm{Tm}_l\ \mathsf{A}) \longrightarrow \mathrm{Tm}_{l'}\ (\mathsf{B}\ \mathsf{u}) && \text{for } (l, l', l'') \in \mathscr{R} \\
&@_{l,l'}\ \mathsf{A}\ \mathsf{B}\ (\lambda_{l,l'}\ \mathsf{A}'\ \mathsf{B}'\ \mathsf{t})\ \mathsf{u} \longmapsto \mathsf{t}\ \mathsf{u} && \text{for } (l, l', l'') \in \mathscr{R}
\end{aligned}$$

Remark 5.2. We stress that there is nothing novel going on: the above declarations are just what one gets when mechanically translating the typing rules and equations defining dependent functions into a logical framework. Indeed, this is the way dependent functions have been defined in equational logical frameworks all along [Ste19, Har21a, Uem21, Ste22b], except (surprisingly) in the Dedukti literature [CD07, BDG$^+$23]. □

These declarations conclude the definition of the theory $\mathbb{T}_{\mathscr{S}}$ for a given finite and functional specification $\mathscr{S}$, which we resume in Figure 5.2 for ease of referencing.

Remark 5.3. Because in our definition of Dedukti we only consider finite theories, the theory of Figure 5.2 is only well-defined when the specification $\mathscr{S} = (\mathscr{L}, \mathscr{A}, \mathscr{R})$ is finite, that is, when the set of levels is finite.[8] Nevertheless, for every term typed in an infinite specification, it is always possible to find a finite fragment of the specification in which this term is still well-typed. Therefore, in some situations it can be acceptable to abuse the notations and write $\mathbb{T}_{\mathscr{S}}$ even when $\mathscr{S}$ is infinite, which then means $\mathbb{T}_{\mathscr{S}'}$ for some finite fragment $\mathscr{S}' \subseteq \mathscr{S}$ large enough for all terms under consideration. Alternatively, under some hypotheses on the specification, one can obtain a finite theory by internalizing the universe levels and indexing the symbols of Figure 5.2 *inside* of the framework [Ass15, Ste19]. This approach will be further developed in Part III, in which we moreover consider a form of *universe polymorphism* similar to the one of Agda. □

[8] This limitation is also present in the encoding of Cousineau and Dowek [CD07], whose definition of Dedukti also forbids infinite theories, yet they actually never discuss it explicitly. Alternatively, as discussed in Section 2.2, we could consider a definition of Dedukti allowing for infinite theories, as used in some other works [Bla20, BGH19].

$$
\begin{array}{ll}
\mathrm{Ty}_{l_\top} : \mathsf{Type} & \text{for } l_\top \in \mathscr{L}_\top \\
\mathrm{Tm}_{l_\top} : \mathrm{Ty}_{l_\top} \longrightarrow \mathsf{Type} & \text{for } l_\top \in \mathscr{L}_\top \\
\mathrm{U}_l : \mathrm{Ty}_{l^+} & \text{for } l \in \mathscr{L} \\
\mathrm{Tm}_{l^+}\ \mathrm{U}_l \longmapsto \mathrm{Ty}_l & \text{for } l \in \mathscr{L} \\
\Pi_{l,l'} : (\mathsf{A} : \mathrm{Ty}_l) \longrightarrow (\mathsf{B} : \mathrm{Tm}_l\ \mathsf{A} \longrightarrow \mathrm{Ty}_{l'}) \longrightarrow \mathrm{Ty}_{l''} & \text{for } (l, l', l'') \in \mathscr{R} \\
\lambda_{l,l'} : (\mathsf{A} : \mathrm{Ty}_l) \longrightarrow (\mathsf{B} : \mathrm{Tm}_l\ \mathsf{A} \longrightarrow \mathrm{Ty}_{l'}) \longrightarrow & \\
\quad ((x : \mathrm{Tm}_l\ \mathsf{A}) \longrightarrow \mathrm{Tm}_{l'}\ (\mathsf{B}\ x)) \longrightarrow \mathrm{Tm}_{l''}\ (\Pi_{l,l'}\ A\ B) & \text{for } (l, l', l'') \in \mathscr{R} \\
@_{l,l'} : (\mathsf{A} : \mathrm{Ty}_l) \longrightarrow (\mathsf{B} : \mathrm{Tm}_l\ \mathsf{A} \longrightarrow \mathrm{Ty}_{l'}) \longrightarrow & \\
\quad (\mathsf{t} : \mathrm{Tm}_{l''}\ (\Pi_{l,l'}\ \mathsf{A}\ \mathsf{B})) \longrightarrow (\mathsf{u} : \mathrm{Tm}_l\ \mathsf{A}) \longrightarrow \mathrm{Tm}_{l'}\ (\mathsf{B}\ \mathsf{u}) & \text{for } (l, l', l'') \in \mathscr{R} \\
@_{l,l'}\ \mathsf{A}\ \mathsf{B}\ (\lambda_{l,l'}\ \mathsf{A}'\ \mathsf{B}'\ \mathsf{t})\ \mathsf{u} \longmapsto \mathsf{t}\ \mathsf{u} & \text{for } (l, l', l'') \in \mathscr{R}
\end{array}
$$

Figure 5.2: Theory $\mathbb{T}_{\mathscr{S}}$ defined by a finite and functional specification $\mathscr{S} = (\mathscr{L}, \mathscr{A}, \mathscr{R})$

The translation function

To conclude the specification of our encoding, we only need to specify its associated translation function $[\![-]\!]$. From the above discussion, it should be clear how to define $[\![-]\!]$: we just map each syntactic constructor of the PTS syntax to its corresponding symbol in the framework, and map variables to variables.

$$
\begin{aligned}
&[\![-]\!] : \mathsf{Tm}_{\mathsf{PTS}} \longrightarrow \mathsf{Tm} \\
&[\![x]\!] := x \\
&[\![\mathsf{U}_l]\!] := \mathrm{U}_l \\
&[\![\Pi_{l,l'} x : A.B]\!] := \Pi_{l,l'}\ [\![A]\!]\ (x.[\![B]\!]) \\
&[\![\lambda^{x:A.B}_{l,l'} x.t]\!] := \lambda_{l,l'}\ [\![A]\!]\ (x.[\![B]\!])\ (x.[\![t]\!]) \\
&[\![t @^{x:A.B}_{l,l'} u]\!] := @_{l,l'}\ [\![A]\!]\ (x.[\![B]\!])\ [\![t]\!]\ [\![u]\!]
\end{aligned}
$$

We can then easily extend the translation function to well-typed contexts: we use the fact that, for Γ well-typed, $x : A \in \Gamma$ implies $\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l$ for some $l \in \mathscr{L}$ (which is unique because the PTS specification is functional) to translate each entry $x : A$ as $x : \mathrm{Tm}_l\ [\![A]\!]$.

$$
\begin{array}{ll}
[\![-]\!] : (\Gamma \in \mathsf{Ctx}_{\mathsf{PTS}}) \longrightarrow \mathsf{Ctx} & \text{such that } \Gamma \vdash_{\mathsf{PTS}} \\
[\![\cdot]\!] := \cdot & \\
[\![\Gamma, x : A]\!] := [\![\Gamma]\!], x : \mathrm{Tm}_l\ [\![A]\!] & \text{where } \Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l
\end{array}
$$

Remark 5.4. As one can see, the fact that we can define $[\![-]\!]$ directly crucially relies on our use of fully annotated PTSs. Had we used the usual PTS syntax, we could follow

Cousineau and Dowek and try to define the translation *only* for the well-typed terms, which would allow us to recover the level annotations l, l' when translating $\Pi x : A.B$ and $\lambda x : A.t$ and $t\ u$. But if we try to use the same trick to recover (for instance) B when translating $\lambda x : A.t$, we would have to define $[\![\lambda x : A.t]\!] := \lambda_{l,l'}\ [\![A]\!]\ (x.[\![B]\!])\ (x.[\![t]\!])$ for B satisfying $\Gamma, x : A \vdash t : B$, yet B is neither unique nor structurally smaller than $\lambda x : A.t$, so this would be ill-defined. In this case, instead of defining $[\![-]\!]$ by structural induction on well-typed terms (which, as just explained, does not work), we could define it by induction on their typing derivations, making it possible to apply a recursive call on B. Yet, because a typing judgment can have multiple derivations, we would then have to show that these are still mapped to convertible Dedukti terms. An alternative approach, taken recently by the author in joint work with Winterhalter [FW24], would be instead to state and prove soundness and conservativity in terms of an inverse translation function, which can be easily defined by structural induction. The direct translation function, which would never be defined explicitly, could then be extracted from the soundness proof. □

Basic properties of the theory

Finally, we conclude this section by showing some basic properties of the theory, which will be essential when proving soundness and conservativity of the encoding.

Notation 5.1. Throughout the rest of this chapter, we write $\Gamma \vdash t : T$ for the Dedukti typing judgment in the theory $\mathbb{T}_{\mathcal{S}}$, where $\mathcal{S}$ is a fixed finite and functional PTS specification. The corresponding PTS judgment in $\mathcal{S}$ is then written $\Gamma \vdash_{\mathrm{PTS}} t : A$. □

Proposition 5.2 (Basic properties of $\mathbb{T}_{\mathcal{S}}$).

(i) **Confluence**: *The rewrite system $\beta\mathcal{R}_{\mathbb{T}_{\mathcal{S}}}$ is confluent.*

(ii) **Well-typedness**: *The theory $\mathbb{T}_{\mathcal{S}}$ is well-typed.*

(iii) **Subject reduction of β**: *If $\Gamma \vdash t : T$ and $t \longrightarrow_{\beta} t'$ then $\Gamma \vdash t' : T$.*

(iv) **Strong normalization of β**: *If $\Gamma \vdash t : T$ then t is strongly normalizing for β.*

Proof. Point (i) follows from the fact that the rewrite system is orthogonal and from Mayr and Nipkow's confluence criterion for orthogonal systems [MN98][9], (ii) can be checked by a routine verification, (iii) follows from Proposition 2.1 and using confluence, and finally (iv) follows from Theorem 4.2 using confluence, well-typedness of $\mathbb{T}_{\mathcal{S}}$ and the fact that there are no arrow-producing rules. ■

Remark 5.5. We could also show subject reduction for $\mathcal{R}_{\mathbb{T}_{\mathcal{S}}}$, however as we will see our proof technique never needs to use this property. □

[9] Note that, although this criterion was shown for the specific rewrite formalism of *Higher-order Rewrite Systems (HRSs)*, following Saillard [Sai15a, Definition 5.2] we can encode the formalism used in Dedukti as a specific HRS, allowing us to use this result in our setting. Alternatively, we refer to Férey's PhD thesis [Fer21], which revisits classic confluence criteria in the rewrite formalism used in Dedukti.

5.3 Soundness

Now that we have seen the definition of our encoding, let us show that it is indeed correct, starting with soundness. First recall that, because we are in a dependently typed setting, typing makes reference to definitional equality, and so the first step to show soundness is to prove that $[\![-]\!]$ respects it. But because definitional equality is defined in terms of rewriting, which in turn uses substitution in its definition, we first have to show that $[\![-]\!]$ also respects substitution and reduction.

Lemma 5.1 (Basic properties of $[\![-]\!]$). *Let $t, u \in \mathsf{Tm}_{\mathsf{PTS}}$.*

1. *We have $[\![t[u/x]]\!] = [\![t]\!][[\![u]\!]/x]$.*
2. *If $t \longrightarrow_\beta u$ then $[\![t]\!] \longrightarrow^* [\![u]\!]$.*
3. *If $t \equiv_\beta u$ then $[\![t]\!] \equiv [\![u]\!]$.*

Proof. The first point follows by induction on t, whereas the second follows by induction on the rewrite position, using the first point for the base case. Finally, the third point follows by induction on $\equiv$ and uses the second point. ■

We can now move to the proof of soundness.

Theorem 5.2 (Soundness). *If $\Gamma \vdash_{\mathsf{PTS}} t : A$ then $[\![\Gamma]\!] \vdash [\![t]\!] : \mathrm{Tm}_{l_\top}\ [\![A]\!]$ for some $l_\top \in \mathscr{L}_\top$.*

Proof. In order for the proof to go through, we instead show the following:

- If $\Gamma \vdash_{\mathsf{PTS}}$ then $[\![\Gamma]\!] \vdash$.
- If $\Gamma \vdash_{\mathsf{PTS}} t : A$ then either $[\![\Gamma]\!] \vdash [\![t]\!] : \mathrm{Tm}_l\ [\![A]\!]$ for some l with $\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l$, or $[\![\Gamma]\!] \vdash [\![t]\!] : \mathrm{Tm}_\top\ [\![A]\!]$ and $A = \mathsf{U}_l$ for some maximal level l.

Observe in the second point that, if A is of the form U_l, then the conclusion is equivalent to $[\![\Gamma]\!] \vdash [\![t]\!] : \mathrm{Tm}_{l^+}\ \mathrm{U}_l$. Moreover, if we already know that $\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l$ then, by uniqueness of levels in functional PTSs, the conclusion implies $[\![\Gamma]\!] \vdash [\![t]\!] : \mathrm{Tm}_l\ [\![A]\!]$. We implicitly use these facts in the proof, which is done by induction on the typing derivation.

- Case EmptyCtx.

$$\frac{}{\cdot \vdash_{\mathsf{PTS}}}$$

Trivial.

- Case ExtCtx.

$$l \in \mathscr{L}\ \frac{\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l}{\Gamma, x : A \vdash_{\mathsf{PTS}}}$$

By i.h., $\llbracket \Gamma \rrbracket \vdash \llbracket A \rrbracket : \mathsf{Tm}_{l^+}\ \mathsf{U}_l$, so $\llbracket \Gamma \rrbracket \vdash \mathsf{Tm}_l\ \llbracket A \rrbracket : \mathsf{Type}$ and $\llbracket \Gamma \rrbracket, x : \mathsf{Tm}_l\ \llbracket A \rrbracket \vdash$.

- Case Var.

$$x : A \in \Gamma\ \frac{\Gamma \vdash_{\mathsf{PTS}}}{\Gamma \vdash_{\mathsf{PTS}} x : A}$$

By i.h. we have $\llbracket \Gamma \rrbracket \vdash$, and moreover $x : A \in \Gamma$ implies $x : \mathsf{Tm}_l\ \llbracket A \rrbracket \in \llbracket \Gamma \rrbracket$ for some l and $\Gamma' \sqsubseteq \Gamma$ with $\Gamma' \vdash_{\mathsf{PTS}} A : \mathsf{U}_l$. By weakening for PTS we have $\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_l$ and by the variable rule in Dedukti we get $\llbracket \Gamma \rrbracket \vdash x : \mathsf{Tm}_l\ \llbracket A \rrbracket$.

- Case Univ.

$$(l_1, l_2) \in \mathscr{A}\ \frac{\Gamma \vdash_{\mathsf{PTS}}}{\Gamma \vdash_{\mathsf{PTS}} \mathsf{U}_{l_1} : \mathsf{U}_{l_2}}$$

By i.h. we have $\llbracket \Gamma \rrbracket$, so we have $\llbracket \Gamma \rrbracket \vdash \mathsf{U}_{l_1} : \mathsf{Ty}_{l_2}$ and hence $\llbracket \Gamma \rrbracket \vdash \mathsf{U}_{l_1} : \mathsf{Tm}_{l_2^+}\ \mathsf{U}_{l_2}$ by the conversion rule.

- Case Pi.

$$(l_1, l_2, l_3) \in \mathscr{R}\ \frac{\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_{l_1} \qquad \Gamma, x : A \vdash_{\mathsf{PTS}} B : \mathsf{U}_{l_2}}{\Gamma \vdash_{\mathsf{PTS}} \Pi_{l_1,l_2} x : A.B : \mathsf{U}_{l_3}}$$

By i.h. we have $\llbracket \Gamma \rrbracket \vdash \llbracket A \rrbracket : \mathsf{Tm}_{l_1^+}\ \mathsf{U}_{l_1}$ and $\llbracket \Gamma \rrbracket, x : \mathsf{Tm}_{l_1}\ \llbracket A \rrbracket \vdash \llbracket B \rrbracket : \mathsf{Tm}_{l_2^+}\ \mathsf{U}_{l_2}$, and thus $\llbracket \Gamma \rrbracket \vdash \llbracket A \rrbracket : \mathsf{Ty}_{l_1}$ and $\llbracket \Gamma \rrbracket \vdash x.\llbracket B \rrbracket : \mathsf{Tm}_{l_1}\ \llbracket A \rrbracket \to \mathsf{Ty}_{l_2}$. Therefore, we have $\llbracket \Gamma \rrbracket \vdash \Pi_{l_1,l_2}\ \llbracket A \rrbracket\ (x.\llbracket B \rrbracket) : \mathsf{Ty}_{l_3}$, and we conclude by applying conversion with $\mathsf{Ty}_{l_3} \equiv \mathsf{Tm}_{l_3^+}\ \mathsf{U}_{l_3}$.

- Case Abs.

$$(l_1, l_2, l_3) \in \mathscr{R}\ \frac{\Gamma \vdash_{\mathsf{PTS}} A : \mathsf{U}_{l_1} \qquad \Gamma, x : A \vdash_{\mathsf{PTS}} B : \mathsf{U}_{l_2} \qquad \Gamma, x : A \vdash_{\mathsf{PTS}} t : B}{\Gamma \vdash_{\mathsf{PTS}} \lambda^{x:A.B}_{l_1,l_2} x.t : \Pi_{l_1,l_2} x : A.B}$$

By i.h. we have $\llbracket \Gamma \rrbracket \vdash \llbracket A \rrbracket : \mathsf{Tm}_{l_1^+}\ \mathsf{U}_{l_1}$ and $\llbracket \Gamma \rrbracket, x : \mathsf{Tm}_{l_1}\ \llbracket A \rrbracket \vdash \llbracket B \rrbracket : \mathsf{Tm}_{l_2^+}\ \mathsf{U}_{l_2}$ and $\llbracket \Gamma \rrbracket, x : \mathsf{Tm}_{l_1}\ \llbracket A \rrbracket \vdash \llbracket t \rrbracket : \mathsf{Tm}_{l_2}\ \llbracket B \rrbracket$. We conclude by deriving $\llbracket \Gamma \rrbracket \vdash \lambda_{l_1,l_2}\ \llbracket A \rrbracket\ (x.\llbracket B \rrbracket)\ (x.\llbracket t \rrbracket) : \mathsf{Tm}_{l_3}\ (\Pi_{l_1,l_2}\ \llbracket A \rrbracket\ (x.\llbracket B \rrbracket))$, and we indeed have $\Gamma \vdash_{\mathsf{PTS}} \Pi_{l_1,l_2} x : A.B : \mathsf{U}_{l_3}$.

- Case App.

$$(l_1,l_2,l_3)\in\mathscr{R}\ \dfrac{\begin{array}{c}\Gamma\vdash_{\mathsf{PTS}} A:\mathsf{U}_{l_1} \qquad \Gamma,x:A\vdash_{\mathsf{PTS}} B:\mathsf{U}_{l_2}\\ \Gamma\vdash_{\mathsf{PTS}} t:\Pi_{l_1,l_2}x:A.B \qquad \Gamma\vdash_{\mathsf{PTS}} u:A\end{array}}{\Gamma\vdash_{\mathsf{PTS}} t@^{x:A.B}_{l_1,l_2}u:B[u/x]}$$

  By i.h. we have $[\![\Gamma]\!]\vdash[\![A]\!]:\mathrm{Tm}_{l_1^+}\ \mathrm{U}_{l_1}$ and $[\![\Gamma]\!],x:\mathrm{Tm}_{l_1}\ [\![A]\!]\vdash[\![B]\!]:\mathrm{Tm}_{l_2^+}\ \mathrm{U}_{l_2}$ and $[\![\Gamma]\!]\vdash[\![u]\!]:\mathrm{Tm}_{l_1}\ [\![A]\!]$. Moreover, we can show $\Gamma\vdash_{\mathsf{PTS}}\Pi_{l_1,l_2}x:A.B:\mathsf{U}_{l_3}$, so by i.h. we also have $[\![\Gamma]\!]\vdash[\![t]\!]:\mathrm{Tm}_{l_3}\ (\Pi_{l_1,l_2}\ [\![A]\!]\ (x.[\![B]\!]))$. Therefore, we obtain $[\![\Gamma]\!]\vdash @_{l_1,l_2}\ [\![A]\!]\ (x.[\![B]\!])\ [\![t]\!]\ [\![u]\!]:\mathrm{Tm}_{l_2}\ ((x.[\![B]\!])[\![u]\!])$, and by applying Lemma 5.1 we get $(x.[\![B]\!])[\![u]\!]\equiv[\![B]\!][[\![u]\!]/x]=[\![B[u/x]]\!]$. We conclude by applying conversion and noting that, by the substitution property for PTSs, we have $\Gamma\vdash_{\mathsf{PTS}} B[u/x]:\mathsf{U}_{l_2}$.

- Case Conv.

$$\begin{array}{c} l\in\mathscr{L}\\ A\equiv_\beta B\end{array}\ \dfrac{\Gamma\vdash_{\mathsf{PTS}} t:A \qquad \Gamma\vdash_{\mathsf{PTS}} B:\mathsf{U}_l}{\Gamma\vdash_{\mathsf{PTS}} t:B}$$

  By i.h. we have $[\![\Gamma]\!]\vdash[\![B]\!]:\mathrm{Tm}_{l^+}\ \mathrm{U}_l$ and thus $[\![\Gamma]\!]\vdash\mathrm{Tm}_l\ [\![B]\!]:\mathrm{Type}$. Moreover, by tedious calculations with Proposition 5.1 we can show $\Gamma\vdash_{\mathsf{PTS}} A:\mathsf{U}_l$, so by the i.h. again we get $[\![\Gamma]\!]\vdash[\![t]\!]:\mathrm{Tm}_l\ [\![A]\!]$. Finally, by Lemma 5.1 we have $[\![A]\!]\equiv[\![B]\!]$, so we conclude by applying conversion to $[\![\Gamma]\!]\vdash[\![t]\!]:\mathrm{Tm}_l\ [\![A]\!]$ with $\mathrm{Tm}_l\ [\![A]\!]\equiv\mathrm{Tm}_l\ [\![B]\!]$. ■

5.4 Conservativity

We now move to the proof of conservativity. As anticipated in Chapter 3, our proof works by taking a term t witnessing $[\![\Gamma]\!]\vdash t:\mathrm{Tm}_l\ [\![A]\!]$, then β-normalizing it (which we can, by Proposition 5.2) and translating it back to the object-language. However, we oversimplified a bit when saying that all β-normal terms can be directly translated back, as illustrated by the following example.

Example 5.1. Consider the functional PTS specification given by $\mathscr{L}=\{0,1,2\}$, and with $\mathscr{A}$ and $\mathscr{R}$ being the graphs of the successor and max functions. Defining the aliases $t:=\lambda_{1,1}\ \mathrm{U}_0\ (x.\mathrm{U}_0)\ (x.x)$ and $u:=@_{1,1}\ \mathrm{U}_0\ (x.\mathrm{U}_0)\ t$, then in the Dedukti encoding we have a derivation of $\cdot\vdash\Pi_{1,0}\ \mathrm{U}_0\ u:\mathrm{Tm}_2\ \mathrm{U}_1$, yet this term is not in the image of $[\![-]\!]$. Indeed, the symbol $@_{1,1}$ is only applied to three arguments, whereas all occurences of this symbol in the image of $[\![-]\!]$ are followed by four arguments. □

The cause for the problem illustrated in this example is the fact that the term considered is not η-long. As explained by Harper *et al.* [HHP93], in logical frameworks, a perfect correspondence with object-language terms can in general only be obtained when considering β-normal η-long forms, also known as *canonical forms*. In frameworks like the

$$
\begin{array}{ll}
|x| := x & |\lambda_{l,l'}\ A\ B\ t| := \lambda_{l,l'}^{x' : |A|.|B\ x'|} x'.|t\ x'| \\
|\mathsf{U}_l| := \mathsf{U}_l & |@_{l,l'}\ A\ B\ t\ u| := |t| @_{l,l'}^{x' : |A|.|B\ x'|} |u| \\
|\Pi_{l,l'}\ A\ B| := \Pi_{l,l'} x' : |A|.|B\ x'| & |(x.t)u| := |t|[|u|/x]
\end{array}
$$

where the x' are fresh

Figure 5.3: (Partial) back-translation function

Edinburgh Logical Framework this is not a problem, because there one includes η-equality in the definition of the framework. This then allows one to show that every well-typed term is $\beta\eta$-equal to a canonical form of the same type [HP05].

Unfortunately, in the case of Dedukti, η-equality is not included in the conversion as it can behave badly with rewriting, and because of its absence it is not true anymore that every term can be mapped to a canonical form of the same type. In order to address this problem, Cousineau and Dowek [CD07] proposed to prove conservativity of β-normal forms in two steps. First, they showed that every β-normal form t witnessing $[\![\Gamma]\!] \vdash t : \mathrm{Tm}_l\ [\![A]\!]$ could be η-expanded while preserving typing, reducing conservativity of β-normal forms to conservativity of canonical forms. They then proceed to show that all canonical forms t witnessing $[\![\Gamma]\!] \vdash t : \mathrm{Tm}_l\ [\![A]\!]$ could be translated back.

In this section, we propose an alternative which allows us to show conservativity more directly. Our main insight is that terms can instead be η-expanded *at (back-)translation time*: if we write $|-|$ for the back-translation function, then $|\Pi_{l,l'}\ A\ B|$ should be the same as $|\Pi_{l,l'}\ A\ (x.B\ x)|$, which should of course be defined as $\Pi_{l,l'} x : |A|.|B\ x|$. However, if B was already an abstraction then by writing $B\ x$ we have just created a β-redex, and thus the back-translation function must also reduce the created redexes at translation time. Moreover, because $B\ x$ is not a strict subterm of $\Pi_{l,l'}\ A\ B$, we cannot simply define $|-|$ by induction on the term. We instead define the back-translation function by induction on the number of symbols and abstractions, by the clauses of Figure 5.3.

As $|-|$ is only partially defined, we call the Dedukti terms in its domain *invertible*.[10] We extend $|-|$ naturally to contexts Γ whose entries are of the form $x : \mathrm{Tm}_l\ A$ with A invertible, in which case Γ is also said to be invertible. As one would expect, $|-|$ is a left-inverse of $[\![-]\!]$. However, because of the η-expansions and β-reductions in the definition of $|-|$, the function $|-|$ is only a right-inverse of $[\![-]\!]$ up to $\beta\eta$-equality.

Proposition 5.3. *We have* $|[\![t]\!]| = t$ *for all* $t \in \mathsf{Tm}_{\mathsf{PTS}}$ *and* $[\![|t|]\!] \equiv_{\beta\eta} t$ *for all* t *invertible.*

Proof. By induction on the definitions of $[\![-]\!]$ and $|-|$. ∎

[10] An explicit description of invertible terms could be of course given, though we do not believe this would be particularly insightful.

Another key property of $|-|$ is that it preserves reduction, and thus conversion. Note that for proving this it is essential that $|-|$ allows for regular β-redexes $(x.t)u$, even though the back-translation can only introduce redexes of the form $(x.t)y$ — also called *β_0-redexes.*

Lemma 5.2 ($|-|$ preserves reduction and conversion).

1. *If t and u are invertible then $t[u/x]$ is invertible and $|t[u/x]| = |t|[|u|/x]$*
2. *If t is invertible and $t \longrightarrow t'$ then t' is invertible and either $|t| \longrightarrow |t'|$ or $|t| = |t'|$.*
3. *If t and u are invertible and $t \equiv u$ then $|t| \equiv |u|$.*

Proof. The first two points are shown by induction on the definition of $|-|$, and the third point follows from the second by using confluence (Proposition 5.2.(i)). ■

Finally, before showing conservativity we will also need some easy technical lemmas. The following first lemma will enable us to show conservativity by induction on β-normal forms, even though $|-|$ might introduce redexes of the form $(y.u)x$. Indeed, among other things, it also proves that such redexes can be reduced without changing the value of the term with respect to $|-|$.

Lemma 5.3. *Given a* Dedukti *term t and variable x', define $\tilde{t} :=$ if $t = x.u$ then $u[x'/x]$ else $t\,x'$.*

1. *If t is β-normal, then $\tilde{t}$ also is.*
2. *If $\tilde{t}$ is invertible, then $t\,x'$ also is and $|\tilde{t}| = |t\,x'|$.*
3. *If $\Gamma \vdash t : (x : T) \to U$ then $\Gamma, x' : T \vdash \tilde{t} : U[x'/x]$.*

Proof. The first point is trivial. For the second point, if t is not an abstraction, then $\tilde{t} = t\,x'$ and the result is trivial, otherwise we have $t = x.u$ and $\tilde{t} = u[x'/x]$, in which case the result easily follows by Lemma 5.2. Finally, the third point follows from tedious calculations, using Proposition 2.1. ■

The following two lemmas then allow us to back-translate conversions from the framework to the object-theory.

Lemma 5.4. *If $\mathrm{Tm}_{l_\top}\ A \equiv \mathrm{Ty}_{l'_\top}$ with $l_\top, l'_\top \in \mathscr{L}_\top$ and A invertible then $l'_\top \neq \top$ and $|A| \equiv \mathrm{U}_{l'_\top}$.*

Proof. By confluence we have $\mathrm{Tm}_{l_\top}\ A \longrightarrow^* \mathrm{Ty}_{l'_\top}$ and thus $A \longrightarrow^* \mathrm{U}_{l'_\top}$, implying in particular that $l'_\top \neq \top$. By Lemma 5.2 with $A \equiv \mathrm{U}_{l'_\top}$ we then conclude $|A| \equiv \mathrm{U}_{l'_\top}$. ■

Lemma 5.5. *If $\mathrm{Tm}_{l_\top}\ A \equiv \mathrm{Tm}_{l'_\top}\ A$ with $l_\top, l'_\top \in \mathscr{L}_\top$ and A, A' invertible then $|A| \equiv |A'|$.*

Proof. By confluence we have $\mathrm{Tm}_{l_\top}\ A \longrightarrow B \ {}^*\!\longleftarrow \mathrm{Tm}_{l'_\top}\ A'$. We then have either $B = \mathrm{Ty}_{l''}$, in which case we conclude by Lemma 5.4, or $B = \mathrm{Tm}_{l''_\top}\ A''$, in which case we must have $l_\top = l'_\top = l''_\top$ and $A \longrightarrow^* A'' \ {}^*\!\longleftarrow A'$, implying $|A| \equiv |A'|$ by Lemma 5.2. ■

We are now ready to show conservativity.

Theorem 5.3 (Conservativity). *If* $\Gamma \vdash_{\mathsf{PTS}} A$ type *and* $[\![\Gamma]\!] \vdash t : \mathrm{Tm}_{l_\top}\ [\![A]\!]$ *for some* $l_\top \in \mathscr{L}_\top$*, then for some* $t' \in \mathsf{Tm}_{\mathsf{PTS}}$ *we have* $t \equiv_{\beta\eta} [\![t']\!]$ *and* $\Gamma \vdash_{\mathsf{PTS}} t' : A$*.*

Proof. By subject reduction and strong normalization of β-reduction (points (iii) and (iv) of Proposition 5.2), it suffices to consider t in β-normal form. We then show the following stronger claim, which implies the result by Proposition 5.3.

Claim 5.1. *Suppose* $|\Gamma| \vdash_{\mathsf{PTS}} |C|$ type *and* $\Gamma \vdash t : \mathrm{Tm}_{l_\top}\ C$ *for some t β-normal. Then t is invertible and* $|\Gamma| \vdash_{\mathsf{PTS}} |t| : |C|$

We prove the claim by induction on the number of symbols in t. First note that cases $t = x.u$ and $t = (x : T_1) \longrightarrow T_2$ and $t =$ Type are all impossible, as by confluence we cannot have $\mathrm{Tm}_{l_\top}\ C$ convertible to a function type or to Type or Kind. Therefore, the only possible case is $t = h\,\vec{u}$ with h a variable or symbol. Moreover, if h is a symbol f, because $\mathrm{Tm}_{l_\top}\ C$ is not convertible to a function type, it follows that f must be fully applied. It is also easy to see that the only candidates for f are U_l, $\Pi_{l,l'}$, $@_{l,l'}$ and $\lambda_{l,l'}$. Finally, if h is a variable, then we must have $\vec{u}$ empty because no variable in the context has a type convertible to a function type.

From all these observations, it thus suffices to consider the following cases. Throughout the rest of the proof, we implicitly use extended conversion from Proposition 5.1 and inversion of symbol application from Proposition 2.1.

- Case $t = x$. By inversion on $\Gamma \vdash x : \mathrm{Tm}_{l_\top}\ C$ we get $x : \mathrm{Tm}_{l'}\ A \in \Gamma$ with $\mathrm{Tm}_{l_\top}\ C \equiv \mathrm{Tm}_{l'}\ A$, which by Lemma 5.5 implies $|A| \equiv |C|$. From $|\Gamma| \vdash_{\mathsf{PTS}}$ we get $|\Gamma| \vdash_{\mathsf{PTS}} x : |A|$, so we conclude $|\Gamma| \vdash_{\mathsf{PTS}} x : |C|$ by applying conversion.

- Case $t = \mathrm{U}_{l_1}$. We have $\mathrm{Tm}_{l_\top}\ C \equiv \mathrm{Ty}_{l_1^+}$, so by Lemma 5.4 we get $l_1^+ \neq \top$ and $|C| \equiv \mathsf{U}_{l_1^+}$. We can thus show $|\Gamma| \vdash_{\mathsf{PTS}} \mathsf{U}_{l_1} : \mathsf{U}_{l_1^+}$ and then $|\Gamma| \vdash_{\mathsf{PTS}} \mathsf{U}_{l_1} : |C|$ by using conversion.

- Case $t = \Pi_{l_1,l_2}\ A\ B$. We have $(l_1, l_2, l_3) \in \mathscr{R}$ and $\Gamma \vdash A : \mathrm{Ty}_{l_1}$ and $\Gamma \vdash B : \mathrm{Tm}_{l_1}\ A \longrightarrow \mathrm{Ty}_{l_2}$ and $\mathrm{Ty}_{l_3} \equiv \mathrm{Tm}_{l_\top}\ C$. Therefore $\Gamma \vdash A : \mathrm{Tm}_{l_1^+}\ \mathrm{U}_{l_1}$, so by i.h. we get $|\Gamma| \vdash_{\mathsf{PTS}} |A| : \mathsf{U}_{l_1}$, and thus $|\Gamma|, x' : |A| \vdash_{\mathsf{PTS}}$. Defining $\tilde{B}$ as in Lemma 5.3, we have $\Gamma, x' : \mathrm{Tm}_{l_1}\ A \vdash \tilde{B} : \mathrm{Tm}_{l_2^+}\ \mathrm{U}_{l_2}$. The term $\tilde{B}$ is β-normal and has less symbols than t, so by i.h. we get $|\Gamma|, x' : |A| \vdash_{\mathsf{PTS}} |\tilde{B}| : \mathsf{U}_{l_2}$, and by Lemma 5.3 we have $|\tilde{B}| = |B\,x'|$. We can now derive $|\Gamma| \vdash_{\mathsf{PTS}} \Pi_{l_1,l_2} x' : |A|.|B\,x'| : \mathsf{U}_{l_3}$. By applying Lemma 5.4 with $\mathrm{Tm}_{l_\top}\ C \equiv \mathrm{Ty}_{l_3}$ we get $|C| \equiv \mathsf{U}_{l_3}$, so by conversion we conclude $|\Gamma| \vdash_{\mathsf{PTS}} \Pi_{l_1,l_2} x' : |A|.|B\,x'| : |C|$.

- Case $t = \lambda_{l_1,l_2}\ A\ B\ u$. We have $(l_1, l_2, l_3) \in \mathscr{R}$ and $\Gamma \vdash A : \mathrm{Ty}_{l_1}$ and $\Gamma \vdash B : \mathrm{Tm}_{l_1}\ A \longrightarrow \mathrm{Ty}_{l_2}$ and $\Gamma \vdash u : (x : \mathrm{Tm}_{l_1}\ A) \longrightarrow \mathrm{Tm}_{l_2}\ (B\,x)$ and $\mathrm{Tm}_{l_\top}\ C \equiv \mathrm{Tm}_{l_3}\ (\Pi_{l_1,l_2}\ A\ B)$. By the same reasoning as in case $t = \Pi_{l_1,l_2}\ A\ B$ we get $|\Gamma| \vdash_{\mathsf{PTS}} |A| : \mathsf{U}_{l_1}$ and $|\Gamma|, x' : |A| \vdash_{\mathsf{PTS}} |B\,x'| : \mathsf{U}_{l_2}$. Defining $\tilde{u}$ as in Lemma 5.3, we have $\Gamma, x' : \mathrm{Tm}_{l_1}\ A \vdash \tilde{u} : \mathrm{Tm}_{l_2}\ (B\,x')$, so by i.h. we get $|\Gamma|, x' : |A| \vdash_{\mathsf{PTS}} |\tilde{u}| : |B\,x'|$, and we have $|\tilde{u}| = |u\,x'|$. We can now derive $|\Gamma| \vdash_{\mathsf{PTS}} \lambda^{x':|A|.|B\ x'|}_{l_1,l_2} x'.|u\ x'| : (\Pi_{l_1,l_2} x' : |A|.|B\ x'|)$. By applying Lemma 5.5 with $\mathrm{Tm}_{l_\top}\ C \equiv \mathrm{Tm}_{l_3}\ (\Pi_{l_1,l_2}\ A\ B)$ we get $|C| \equiv \Pi_{l_1,l_2} x' : |A|.|B\ x'|$, so we conclude $|\Gamma| \vdash_{\mathsf{PTS}} \lambda^{x':|A|.|B\ x'|}_{l_1,l_2} x'.|u\ x'| : |C|$ with conversion.

- Case $t = @_{l_1,l_2}\ A\ B\ u\ v$. We have $(l_1, l_2, l_3) \in \mathscr{R}$ and $\Gamma \vdash A : \mathrm{U}_{l_1}$ and $\Gamma \vdash B : \mathrm{Tm}_{l_1}\ A \longrightarrow \mathrm{Ty}_{l_2}$ and $\Gamma \vdash u : \mathrm{Tm}_{l_3}\ (\Pi_{l_1,l_2}\ A\ B)$ and $\Gamma \vdash v : \mathrm{Tm}_{l_1}\ A$ and $\mathrm{Tm}_{l_\top}\ C \equiv \mathrm{Tm}_{l_2}\ (B\ v)$. By the same reasoning as in case $t = \Pi_{l_1,l_2}\ A\ B$, we get $|\Gamma| \vdash_{\mathsf{PTS}} |A| : \mathsf{U}_{l_1}$ and $|\Gamma|, x' : |A| \vdash_{\mathsf{PTS}} |B\ x'| : \mathsf{U}_{l_2}$. We can then derive $|\Gamma| \vdash_{\mathsf{PTS}} \Pi_{l_1,l_2} x' : |A|.|B\ x'| : \mathsf{U}_{l_3}$, allowing us to apply the i.h. to $\Gamma \vdash u : \mathrm{Tm}_{l_3}\ (\Pi_{l_1,l_2}\ A\ B)$, yielding $|\Gamma| \vdash_{\mathsf{PTS}} |u| : \Pi_{l_1,l_2} x' : |A|.|B\ x'|$. Then by the i.h. applied to $\Gamma \vdash v : \mathrm{Tm}_{l_1}\ A$ we get $|\Gamma| \vdash_{\mathsf{PTS}} |v| : |A|$, so we get $|\Gamma| \vdash_{\mathsf{PTS}} |u| @^{x':|A|.|B\ x'|}_{l_1,l_2} |v| : |B\ x'|[|v|/x']$. Because x' is fresh, we have $|B\ x'|[|v|/x'] = |B\ v|$ by Lemma 5.2. Moreover, by Lemma 5.5 with $\mathrm{Tm}_{l_\top}\ C \equiv \mathrm{Tm}_{l_2}\ (B\ v)$ we get $|C| \equiv |B\ v|$, so we can conclude $|\Gamma| \vdash_{\mathsf{PTS}} |u| @^{x':|A|.|B\ x'|}_{l_1,l_2} |v| : |C|$ by applying conversion. ■

Part II

Generic Bidirectional Typing in a Logical Framework

Chapter 6

Introduction to Part II

The problem of annotations

In the previous part, we have seen how proofs of conservativity can be made smoother by avoiding arrow-producing rules and applying Theorem 4.2 to reduce general conservativity to conservativity of β-normal forms. This allowed us to give a new encoding of Pure Type Systems in Dedukti and to prove its conservativity in a fairly straightforward manner.

However, if we try to use our encoding to type-check terms in practice in Dedukti, we notice a problem: while an application is most often written as just $t\ u$, in our encoding it is represented in a fully annotated manner, as $@_{l,l'}\ [\![A]\!]\ (x.[\![B]\!])\ [\![t]\!]\ [\![u]\!]$. Because Dedukti is developed specifically with the intent of being used in practice to type-check large proof libraries, it is clear that this increase on the size of terms can be a problem.

The original PTS encoding in Dedukti avoided this problem by instead using the rule $\mathrm{Tm}\ (\Pi\ \mathsf{A}\ \mathsf{B}) \longmapsto (x : \mathrm{Tm}\ \mathsf{A}) \to \mathrm{Tm}\ (\mathsf{B}\ x)$ to represent the object language abstraction and application by the ones of the framework, which are less annotated. However, this technique for avoiding annotations is not at all general: for instance, to encode dependent sums using this strategy, we would need the rule $\mathrm{Tm}\ (\Sigma\ \mathsf{A}\ \mathsf{B}) \longmapsto (x : \mathrm{Tm}\ \mathsf{A}) \times \mathrm{Tm}\ (\mathsf{B}\ x)$, which is of course not valid because Dedukti does not support dependent sums natively. One could of course try to extend the framework with dependent sums to cover the previous example, however this technique then breaks completely for encoding *positive* types, such as lists, given that the rule $\mathrm{Tm}\ (\mathrm{List}\ \mathsf{A}) \longmapsto \mathbf{List}_{\mathrm{Tm}\ \mathsf{A}}$ would not yield the correct implementation. Therefore, for defining full-fledged dependent type theories that go beyond basic type formers, the proliferation of annotations is unfortunately unavoidable in Dedukti.

Towards a framework supporting non-annotated syntaxes

The problem we have just described is actually not at all specific to Dedukti. In general, seeing type theories as theories in a logical framework yields *fully annotated* syntactic presentations, in which all arguments are explicitly spelled out: an application is written as $t@_{A,x.B}u$, a dependent pair as $\langle t, u\rangle_{A,x.B}$, cons as $t\ ::_A\ l$, etc. If we chose to take logical frameworks seriously as a *specification* of what type theories are, as many indeed do [AK16a, Ste22b, Gra23], we then see that the usual presentations of type theories are the ones which deviate in being too economic. There are many reasons which suggest that the fully annotated syntax is the most canonical one: for instance, one of its benefits is that all the subjects of premises in typing rules get recorded in the syntax, as illustrated in the following rule for dependent sums. In contrast, non-annotated syntaxes only chose to record a subset of the subjects, which is arguably a less canonical choice than keeping all of them.

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B \text{ type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t, u \rangle_{A,x.B} : \Sigma x : A.B}$$

However, fully annotated terms are not only much slower to type-check and reduce, but it is also not reasonable to ask users of type theories to write all such annotations. This is also why the syntactic presentations of type theory that we use in practice omit the majority of these annotations, allowing one to write $t\ u$ for application,[1] $\langle t, u \rangle$ for a dependent pair, $t :: l$ for cons, etc. Therefore, even if fully annotated syntaxes might be the most canonical choice, they are just not the best choice for a framework like Dedukti, which is developed with the specific intention of being used in practice as a theory-independent type-checker. It is thus natural to wonder if one could modify Dedukti to support the non-annotated syntaxes we know and love.

Type-checking non-annotated terms

If removing annotations can be very desirable, it is nevertheless important to note that it has a cost: because knowing them is still important when typing terms, it becomes unclear how to type terms algorithmically, even when the definitional equality of the theory is decidable. For instance, if we omit the annotations in $\langle t, u \rangle_{A,x.B}$, then when building a typing derivation for this term one has to guess A and B:

$$\frac{\Gamma \vdash ? \text{ type} \qquad \Gamma, x : ? \vdash ? \text{ type} \qquad \Gamma \vdash t : ? \qquad \Gamma \vdash u : ?}{\Gamma \vdash \langle t, u \rangle : \Sigma x : ?. ?}$$

We thus see that the problem of removing annotations is somewhat dual to the one of type-checking terms. If a variant of Dedukti supporting non-annotated syntaxes is to be implementable in practice, we need to strike a balance between these two points and explain why omitting annotations does not jeopardize decidability of type-checking.

Bidirectional typing

The aforementioned problem is thankfully well known in the literature, and a solution for it is provided by *bidirectional typing* [McB18, Coq96, DK21, DP04, PT00], a typing

[1] This non-annotated syntax is so common that many might not even realize that an omission is being made!

discipline in which the declarative typing judgment $\Gamma \vdash t : A$ is decomposed explicitly into inference $\Gamma \vdash t \Rightarrow A$, where Γ and t are inputs and A is an output, and checking $\Gamma \vdash t \Leftarrow A$, where Γ, t and A are all inputs. The important point is that, by using these new judgments to control the flow of type information in typing rules, one can specify algorithmically how these rules should be used. For instance, the following rule clarifies how one should type $\langle t, u \rangle$: the types A and B are not to be guessed, but instead recovered from the type C, which should be given as input.

$$\frac{C \longrightarrow^* \Sigma x : A.B \qquad \Gamma \vdash t \Leftarrow A \qquad \Gamma \vdash u \Leftarrow B[t/x]}{\Gamma \vdash \langle t, u \rangle \Leftarrow C}$$

In general, whenever a term starts by a *constructor* (that is, an *introduction form*), bidirectional typing allows the recovery of annotations by asking its type to be given as input. Dually to how constructors can be type-checked, bidirectional typing supports type-inference of *destructors* (that is, of *elimination forms*) by recovering the missing arguments from the type of its first argument, which gets inferred. This can be illustrated with the bidirectional typing rule for application:

$$\frac{\Gamma \vdash t \Rightarrow C \qquad C \longrightarrow^* \Pi x : A.B \qquad \Gamma \vdash u \Leftarrow A}{\Gamma \vdash t\, u \Rightarrow B[u/x]}$$

We therefore see that bidirectional typing is the natural companion for an non-annotated syntax, as it allows to algorithmically explain how the missing information can be retrieved.

Our contribution

We start in Chapter 7 by proposing a new logical framework in the same family as Dedukti, where the definitional equality is specified by rewrite rules, but in which we can present theories using the more compact non-annotated syntax we are used to. For instance, we can define dependent functions in our framework by the following theory:[2]

$$\begin{aligned}
&\mathrm{Ty}(\cdot)\ \text{sort}, \quad \mathrm{Tm}(\mathsf{A} : \mathrm{Ty})\ \text{sort}, \quad \Pi(\cdot\,;\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}) : \mathrm{Ty},\\
&\lambda(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\,;\mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\\
&@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\,;\mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}),\\
&@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}
\end{aligned}$$

While we cannot fully explain this example at this point, we would like to highlight how the arguments A and B are separated from the other ones by a separator ; in λ and $@$, which indicates that they are *erased*. This means that an abstraction is then

[2] Slightly abusing the notations we present in Section 7.2.

written as just $\lambda(x.t)$ and an application as just $@(t, u)$, and so we indeed obtain the syntax we wanted. Note the difference with *implicit arguments* [Nor07], which allow for the omission of annotations in the user-level syntax, that are then elaborated through unification when going to the real syntax: in our framework, the arguments A and B are really not there! Therefore, unlike with implicit arguments, erased arguments are *computationally irrelevant*, meaning that the result of a computation cannot depend on them. Consequently, rewriting and equality-checking are performed without taking these arguments into account, which can then be done much more efficiently compared to when fully annotated terms are used.

However, as explained previously, omitting arguments from the syntax can jeopardize the decidability of typing, as we are then led to guess the missing information. We solve this in Chapter 8 by refining the definition of theories of Chapter 7 into *bidirectional theories*. The main change with respect to regular theories is the separation of (term-level) schematic rules as either *constructor* or *destructor* rules, motivated by the aforementioned differences of their roles with regards to bidirectionality. For instance, the theory for dependent functions presented just above fits the format for bidirectional theories by seeing Π and λ as constructors and $@$ as a destructor.

Then, to formulate the bidirectional type system generated by a bidirectional theory, we first address the well-known problem that some non-annotated terms cannot be algorithmically typed — a limitation that is not at all specific to bidirectional typing [Dow93]. This is typically the case for redexes like $@(\lambda(x.t), u)$: because $@$ is a destructor, then we require its first argument to be inferred, but because λ is a constructor, then this means that $\lambda(x.t)$ can only be type-checked. To rule out this issue, we define our bidirectional system over a syntax of *inferable* and *checkable* terms, in which an *ascription* $t :: T$ must be inserted whenever a destructor meets a constructor — similarly to bidirectional typing *à la* McBride [McB18].

We then prove our main results, showing the correctness of the bidirectional type system with respect to the regular one of Chapter 7 (which we sometimes refer to as *declarative*). We first establish *soundness*, ensuring that any term typed by the bidirectional system is also typable by the declarative one when forgetting about ascriptions. Dually, we show *annotability* [DK21], ensuring that any (declaratively) typed term can be sufficiently annotated with ascriptions to get a term typable by the bidirectional system. Finally, we also show that the bidirectional system is decidable for strongly normalizing bidirectional theories, allowing it to be used for typing terms algorithmically.

We would like to highlight that, while bidirectional typing has been fruitfully studied in the setting of various specific theories [Coq96, LB21, GSB19, AA11, Nor07, AC05], its general theory has nevertheless remained mostly informal and not fully developed [McB18, DK21, McB22]. Therefore, a key aspect of our contribution is that, by formulating bidirectional typing in the setting of a logical framework, we give it a *generic* formal treatment for the whole class of type theories that fit our definition of bidirectional theory, putting its principles in solid ground.

Revising Dedukti's design choices

While the main aspects of our proposal are the removal of annotations and the incorporation of bidirectional typing, we also take this opportunity of building a framework from scratch to revise some of Dedukti's design choices. For instance, recall that showing conservativity of our PTS encoding in Chapter 5 was made more difficult by the fact that the non-canonical forms of Dedukti do not correspond directly to terms of the object-language. As an example, the Dedukti term $(zy.z \llbracket A \rrbracket\ (x.\llbracket B \rrbracket)\ y \llbracket u \rrbracket)\ @_{l,l'} \llbracket t \rrbracket$ is not in the image of $\llbracket - \rrbracket$, and we first need to β-normalize it to obtain $@_{l,l'} \llbracket A \rrbracket\ (x.\llbracket B \rrbracket) \llbracket t \rrbracket \llbracket u \rrbracket$, which can now be identified as the translation of $t@^{x:A.B}_{l,l'}u$. In an ideal world, it would be much better to not even have to consider terms like $(zy.z \llbracket A \rrbracket\ (x.\llbracket B \rrbracket)\ y \llbracket u \rrbracket)\ @_{l,l'} \llbracket t \rrbracket$, which would make conservativity proofs much more direct.

To address this inconvenience, our framework's design also eliminates such bureaucratic terms by allowing only for canonical forms, an approach pioneered long ago by *Concurrent LF* [WCPW03] and whose subsequent integration into ELF yielded *Canonical LF* [HL07]. However, while Concurrent and Canonical LF eliminated non-canonical forms with the aim of reducing $\beta\eta$-equality-checking to syntactic equality-checking, our goal in adopting this idea is more akin the goals of Haselwarter and Bauer [HB23], Uemura [Uem21] and Adams [Ada08]. Like ours, their frameworks allow only for terms that directly correspond to object-theory terms, with the objective of bringing closer the presentation of theories in the framework with the usual presentations used in practice. Finally, removing terms like $(xy.x \llbracket A \rrbracket\ (x.\llbracket B \rrbracket)\ y \llbracket u \rrbracket)\ @_{l,l'} \llbracket t \rrbracket$ also seems to be a pre-requisite to support bidirectional typing, after all how can we specify that the head $\llbracket t \rrbracket$ of the application should be inferred to obtain $\llbracket A \rrbracket$ and $\llbracket B \rrbracket$ when $@_{l,l'}$ can appear partially applied?

Following Uemura [Uem21] and Haselwarter and Bauer [HB23] once again, we also restrict our framework to only allow for *second-order* theories, departing from the ELF and Dedukti tradition which allows for theories of arbitrary orders. Even though this limits the theories one can define, as argued by Uemura [Uem21, Remark 3.2.12] most type theories are of order at most two, as when a type theory supports function types these can always be used to lower the order of higher-order symbols.[3] This allows for a cleaner and more straightforward definition of the framework: for instance, the definition of canonical-form-preserving substitution (known as *hereditary substitution* [WCPW03, HL07]) is very technical in a higher-order setting [HL07, Figure 5], whereas we consider our second-order definition of substitution (Figure 7.1) to be much simpler.

Finally, in this part we are also more careful when defining the raw syntax of our framework, by presenting it in an *intrinsically scoped* manner. This means that, instead of talking about a set of terms, we talk about a *family* of them, indexed by *scopes* specifying which variables can appear. This allows us to define operations such as substitution in a

[3] For instance, the eliminator of W types is almost always formulated using function types, which yields a second-order formulation of a symbol which would otherwise be third-order.

much more natural and mathematically rigorous way.

The implementation

The main goal of this part is therefore to allow for the removal of annotations while keeping type-checking algorithmic, thanks to the decidability of the bidirectional system. This decidability result allowed us to implement our framework in the theory-independent type-checker BiTTs, discussed in Chapter 10 and publicly available at

https://github.com/thiagofelicissimo/BiTTs

This has allowed our proposal to be used in practice with multiple theories, from variants of Martin-Löf Type Theory with various type formers and kinds of universes, to Higher-Order Logic and more modern theories like Exceptional Type Theory and Observational Type Theory. These examples are discussed in Chapter 9 and detailed in the implementation.

Because of its support for non-annotated syntaxes, BiTTs can allow for better performances when compared with Dedukti, which makes it a good candidate for cross-checking large libraries of proofs coming from proof assistants. Finally, our implementation can also be used to prototype with new type theories without having to implement a new bidirectional algorithm from scratch, an important commodity given the large number of new theories that are proposed each year.

Related version

A preliminary version of this work had first appeared in the proceedings of the 33rd European Symposium on Programming (ESOP 2024) [Fel24a]. The framework given here is instead closer to an extended version currently under submission for a journal. The improvements are substantial, and contain in particular the following:

1. We have addressed the main deficiency of our preliminary work, which was its lack of support for indexed types, such as vectors and, most importantly, the equality type: our notion of theory has been updated with equational premises in order to support such types.

2. We have extended the bidirectional syntax by adding support for ascriptions, allowing to turn a checkable term t into an inferable one $t :: T$. Accordingly, our notion of completeness has been updated to annotability, as discussed in the introduction. Our previous completeness result, here called *ascription-free completeness*, is now shown as a simple corollary of annotability.

3. The requirement that term-level rules should be classified as constructor and destructor rules does not appear anymore in the definition of the framework, where

this distinction was irrelevant and introduced duplication in some places. Instead, this condition only appears when refining the notion of theory into bidirectional theory, when introducing the bidirectional system in Chapter 8.

4. While our previous work required rewrite rules left-hand sides to be headed by destructors, this restriction is dropped here, enabling the definition of theories whose rules do not satisfied this condition (for instance, type theory with Russell-style universes).

5. Our treatment of matching modulo has been simplified, in particular by removing its reliance on the maximal-outermost strategy. This brings the theory closer to our implementation, where we use a reduction strategy based on call-by-value.

6. The section on examples of theories covered by our framework has been considerably extended, better illustrating the generality of our approach.

Let us however note that the extended version currently under review lags behind the version presented here, in particular by not implementing the improvements discussed in points 3 and 4 — these were only discovered after the submission was made, and will be integrated to it in the next revision.

Chapter 7

A Logical Framework with Erased Arguments

In this chapter we define a logical framework in the same family as Dedukti, but in which we can specify theories with non-annotated syntaxes, by declaring some of their arguments as *erased.* We start by defining the raw syntax of our framework, followed by our definition of theories, which are made of rewrite rules and *schematic typing rules.* These rules then give rise to the type system of a type theory, which is defined subsequently. This system is then used to define the *valid theories*, a refinement of our notion of theory in which typing information is also taken into account. We then conclude the chapter by showing that the type system of a theory satisfies desirable properties, such as weakening and substitution.

7.1 Raw syntax

Intrinsically scoped syntax

When defining the syntax of a type theory, and like we did when defining the one of Dedukti, one most usually defines a set Tm containing all terms. However, it is sometimes useful to distinguish terms $t \in \mathsf{Tm}$ according to the variables that can appear in t — for instance, when defining the application of a substitution, it is useful to know that it is defined for all variables in t. Therefore, we elect an *intrinsically scoped* presentation of syntax, in which we define instead a *family* of terms, indexed by *scopes* specifying which variables can appear.

Scopes and signatures

The basic ingredients of our raw expressions are actually not only *variables* $x, y, \ldots$, but also *metavariables* $\mathsf{x}, \mathsf{y}, \ldots$ and *symbols* $f, g, \ldots$, which are specified respectively by *(variable) scopes*, *metavariable scopes* and *signatures*.

$$\begin{aligned} \boxed{\text{Scope}} \ni \quad & \gamma, \delta ::= \cdot \mid \gamma, x \\ \boxed{\text{MScope}} \ni \quad & \theta, \xi ::= \cdot \mid \theta, \mathsf{x}\{\delta\} \\ \boxed{\text{Sig}} \ni \quad & \Sigma ::= \cdot \mid \Sigma, f(\theta) \end{aligned}$$

A scope γ is simply a list of variables, whereas a metavariable scope θ is a list of metavariables accompanied by a variable scope δ, explaining the arguments each metavariable expects. For instance, $\mathsf{x}\{x, y\} \in \theta$ specifies a metavariable x taking two arguments, each one named by the variables x or y. We have an obvious operation of concatenation $\gamma.\delta$ and $\theta.\xi$, for variable scopes and metavariable scopes respectively. A signature Σ is then a list of symbols accompanied by a metavariable scope explaining its arguments: for instance, $f(\mathsf{x}\{\cdot\}, \mathsf{y}\{x\}) \in \Sigma$ specifies a symbol f taking two arguments and binding one variable in its second argument. We allow ourselves to abbreviate $\mathsf{x}\{\cdot\}$ as x, and $f(\cdot)$ as f when convenient.

Example 7.1. The following signature $\Sigma_{\lambda\Pi}$ defines the raw syntax of a minimalistic Martin-Löf Type Theory (MLTT) with only dependent functions.

$$\mathrm{Ty},\ \mathrm{Tm}(\mathsf{A}),\ \Pi(\mathsf{A}, \mathsf{B}\{x\}),\ \lambda(\mathsf{t}\{x\}),\ @(\mathsf{t}, \mathsf{u}) \qquad (\Sigma_{\lambda\Pi})$$

The symbols are the ones we would expect, except perhaps for Ty and Tm whose role will become clear later. □

Remark 7.1. Here, variables $x, y, \ldots$ correspond to Dedukti's variables of order 0, while metavariables $\mathsf{t}, \mathsf{u}, \ldots$ correspond to Dedukti's variables of order 1, and symbols $f, g, \ldots$ correspond to Dedukti's symbols, which in our framework can be of order at most 2. Note also that our signatures only store information about the raw syntax, and therefore are unrelated to the signatures of the ELF literature, which also store typing information. □

Remark 7.2. When working with de Bruijn indices, a scope γ becomes just a natural number and a metavariable scope θ becomes just a list of natural numbers. In this setting, our definition of signatures corresponds exactly to the well-known *binding-signatures* [Acz78, FPT99], where $f(\mathsf{x}_1\{x^1_1, \ldots, x^1_{k_1}\}, \ldots, \mathsf{x}_n\{x^n_1, \ldots, x^n_{k_n}\})$ is represented by $(k_1, \ldots, k_n)$. □

Terms and substitutions

Given a fixed signature Σ, we define *terms*, *(variable) substitutions* and *metavariable substitutions* by the following grammars. Note that, as explained before, we do not define a set of terms but instead a family $\mathsf{Tm}\ \theta\ \gamma$ in which the indices θ and γ explain which metavariables and variables are in scope.

$$\begin{array}{rrll}
\boxed{\mathsf{Tm}\ \theta\ \gamma} \ni & t, u, T, U ::= & \mid x & \text{if } x \in \gamma \\
& & \mid \mathsf{x}\{\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta\} & \text{if } \mathsf{x}\{\delta\} \in \theta \\
& & \mid f(\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi) & \text{if } f(\xi) \in \Sigma \\
\boxed{\mathsf{Sub}\ \theta\ \gamma\ \delta} \ni & \vec{t}, \vec{u}, \vec{s}, \vec{v} ::= & \mid \varepsilon & \text{if } \delta = \cdot \\
& & \mid \vec{u} \in \mathsf{Sub}\ \theta\ \gamma\ \delta', t \in \mathsf{Tm}\ \theta\ \gamma & \text{if } \delta = \delta', x \\
\boxed{\mathsf{MSub}\ \theta\ \gamma\ \xi} \ni & \mathbf{t}, \mathbf{u}, \mathbf{s}, \mathbf{v} ::= & \mid \varepsilon & \text{if } \xi = \cdot \\
& & \mid \mathbf{u} \in \mathsf{MSub}\ \theta\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}\ \theta\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\}
\end{array}$$

A term is either a variable x, a metavariable x applied to a substitution $\vec{t}$, or a symbol f applied to a metavariable substitution $\mathbf{t}$. A (variable) substitution $\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ is then simply a list of terms, in which each term corresponds to one of the variables in δ. Similarly, a metavariable substitution $\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ is also a list of terms, with the difference that each position $\mathsf{x}\{\delta\} \in \xi$ extends the current scope γ with the variables in δ. We write this variable binding as $\vec{x}_\delta.t$, which can be seen in action in the cases of λ and Π in Example 7.1. Finally, when convenient, we allow ourselves to abbreviate $\mathsf{x}\{\varepsilon\}$ as x and $f(\varepsilon)$ as f.

Example 7.1. The terms defined by the signature $\Sigma_{\lambda\Pi}$ are given by the following grammar, where we omit the scope requirements for variables and metavariables.

$$t, u, A, B ::= x \mid \mathsf{x}\{\vec{t}\} \mid \mathsf{Ty} \mid \mathsf{Tm}(A) \mid \lambda(x.t) \mid \Pi(A, x.B) \mid @(t, u) \qquad \square$$

Given a metavariable substitution $\mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ and $\mathsf{x}\{\delta\} \in \xi$, we write $\mathbf{t}_{\mathsf{x}} \in \mathsf{Tm}\ \theta\ \gamma.\delta$ for the term in $\mathbf{t}$ at the position pointed by x. Similarly, given a substitution $\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ and $x \in \delta$, we write $t_x \in \mathsf{Tm}\ \theta\ \gamma$ for the term in $\vec{t}$ at the position pointed by x.

For each γ and θ we have the *identity substitutions* $\mathsf{id}_\gamma \in \mathsf{Sub}\ (\cdot)\ \gamma\ \gamma$ and $\mathsf{id}_\theta \in \mathsf{MSub}\ \theta\ (\cdot)\ \theta$, defined by $\mathsf{id}_{(\cdot)} := \varepsilon$ and $\mathsf{id}_{\gamma,x} := \mathsf{id}_\gamma, x$ and $\mathsf{id}_{\theta,\mathsf{x}\{\gamma\}} := \mathsf{id}_\theta, \vec{x}_\gamma.\mathsf{x}\{\mathsf{id}_\gamma\}$.[1] While the identity variable substitution id_γ is just the list of variables from γ, the identity metavariable substitution id_θ needs to "eta-expand" each metavariable $\mathsf{x}\{\delta\} \in \theta$ to $\vec{x}_\delta.\mathsf{x}\{\mathsf{id}_\delta\}$ in order for the result to be a valid metavariable substitution. Finally, we allow ourselves to omit the index of id_γ or id_θ when it can be inferred from the context.

Contexts

Given a fixed signature Σ, we define *(variable) contexts* and *metavariable contexts* by the following grammars. These are defined simultaneously[2] with two functions $| - |$ computing their *underlying scopes* $|\Gamma| \in \mathsf{Scope}$ and $|\Theta| \in \mathsf{MScope}$, given by $| \cdot | := \cdot$ and

[1] Note that this definition uses implicit weakenings.

[2] Technically, these are defined by *small induction-recursion* [Dyb00].

$|\Gamma, x : T| := |\Gamma|, x$ and $|\Theta, \mathsf{x}\{\Delta\} : T| := |\Theta|, \mathsf{x}\{|\Delta|\}$.

$$\boxed{\mathsf{Ctx}\ \theta\ \gamma} \ni \qquad \Gamma, \Delta ::= \cdot \mid \Gamma \in \mathsf{Ctx}\ \theta\ \gamma, x : T \in \mathsf{Tm}\ \theta\ \gamma.|\Gamma|$$

$$\boxed{\mathsf{MCtx}\ \theta} \ni \qquad \Theta, \Xi ::= \cdot \mid \Theta \in \mathsf{MCtx}\ \theta, \mathsf{x}\{\Gamma \in \mathsf{Ctx}\ \theta.|\Theta|\ (\cdot)\} : T \in \mathsf{Tm}\ \theta.|\Theta|\ |\Gamma|$$

A context $\Gamma \in \mathsf{Ctx}\ \gamma\ \theta$ is either empty, or composed by a context $\Gamma' \in \mathsf{Ctx}\ \gamma\ \theta$ and a variable x with a term $T \in \mathsf{Tm}\ \theta\ \gamma.|\Gamma'|$. An important point to note is that the term T does not live in scope γ, but in the extension of γ with the underlying scope of Γ'. This means that if a context $x_1 : T_1, \ldots, x_k : T_k$ lives in some scope γ, then each term T_i lives in $\gamma, x_1, \ldots, x_{i-1}$ and thus also has access to the previously occurring variables. The case of a metavariable context $\Theta \in \mathsf{MCtx}\ \theta$ is similar: we have either Θ empty or $\Theta = \Theta', \mathsf{x}\{\Delta\} : T$, where Δ has access to metavariables in θ and Θ, and T has moreover access to the variables in Δ.

Given $\Gamma \in \mathsf{Ctx}\ \theta\ \gamma$ and $\Delta \in \mathsf{Ctx}\ \theta\ \gamma.|\Gamma|$, we write $\Gamma.\Delta \in \mathsf{Ctx}\ \theta\ \gamma$ for their concatenation, defined by induction on Γ. Similarly, given $\Theta \in \mathsf{MCtx}\ \theta$ and $\Xi \in \mathsf{MCtx}\ \theta.|\Theta|$, we write $\Theta.\Xi \in \mathsf{MCtx}\ \theta$ for their concatenation, defined by induction on Ξ.[3]

Notation 7.2. We establish the following notations.

- We write $e \in \mathsf{Expr}\ \theta\ \gamma$ for either $e \in \mathsf{Tm}\ \theta\ \gamma$ or $e \in \mathsf{Sub}\ \theta\ \gamma\ \delta$ or $e \in \mathsf{MSub}\ \theta\ \gamma\ \xi$ or $e \in \mathsf{Ctx}\ \theta\ \gamma$.
- We write Tm_Σ, Sub_Σ, MSub_Σ, Ctx_Σ and MCtx_Σ when Σ is not clear from the context.
- We write $\mathsf{Ctx}\ \theta$ for $\mathsf{Ctx}\ \theta\ (\cdot)$, Ctx for $\mathsf{Ctx}\ (\cdot)\ (\cdot)$ and MCtx for $\mathsf{MCtx}\ (\cdot)$. □

Remark 7.3. We work with a nameful syntax, allowing us to implicitly weaken expressions: if $e \in \mathsf{Expr}\ \theta\ \gamma$ and θ is a subsequence of θ' and γ is a subsequence of γ' then we also have $e \in \mathsf{Expr}\ \theta'\ \gamma'$. Nevertheless, we expect that our proofs can be formally carried out using de Bruijn indices, by properly inserting weakenings whenever needed, and showing the associated lemmata. □

Substitution application

We define in Figure 7.1 the application of a (variable or metavariable) substitution to an expression[4]. Given a variable substitution $\vec{v} \in \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2$ its application to an expression $e \in \mathsf{Expr}\ \theta\ \gamma_2$ gives $e[\vec{v}] \in \mathsf{Expr}\ \theta\ \gamma_1$, and given a metavariable substitution $\mathbf{v} \in \mathsf{MSub}\ \theta_1\ \delta\ \theta_2$ its application to an expression $e \in \mathsf{Expr}\ \theta_2\ \gamma$ gives $e[\mathbf{v}] \in \mathsf{Expr}\ \theta_1\ \delta.\gamma$.

The main case of the definition is when we substitute $\mathbf{v} \in \mathsf{MSub}\ \theta_1\ \delta\ \theta_2$ in the term $\mathsf{x}\{\vec{t}\} \in \mathsf{Tm}\ \theta_2\ \gamma$, where $\mathsf{x}\{\gamma_\mathsf{x}\} \in \theta_2$. By first recursively substituting $\mathbf{v}$ in $\vec{t} \in \mathsf{Sub}\ \theta_2\ \gamma\ \gamma_\mathsf{x}$ we get $\vec{t}[\mathbf{v}] \in \mathsf{Sub}\ \theta_1\ \delta.\gamma\ \gamma_\mathsf{x}$. We moreover have $\mathbf{v}_\mathsf{x} \in \mathsf{Tm}\ \theta_1\ \delta.\gamma_\mathsf{x}$, so by substituting the variables in γ_x by $\vec{t}[\mathbf{v}]$ and the ones in δ by themselves we get $\mathbf{v}_\mathsf{x}[\mathsf{id}, \vec{t}[\mathbf{v}]] \in \mathsf{Tm}\ \theta_1\ \delta.\gamma$ as the final result.

[3] Technically, these must be defined simultaneously with proofs that $|\Gamma.\Delta| = |\Gamma|.|\Delta|$ and $|\Theta.\Xi| = |\Theta|.|\Xi|$, by *recursion-recursion* [Nor13].

[4] Similarly to concatenation, this definition is done by recursion-recursion with proofs that $|\Gamma[\vec{v}]| = |\Gamma|$ and $|\Gamma[\mathbf{v}]| = |\Gamma|$

$$
\begin{aligned}
& - [-] : \mathsf{Tm}\ \theta\ \gamma_2 \longrightarrow \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2 \longrightarrow \mathsf{Tm}\ \theta\ \gamma_1 \\
& x[\vec{v}] := v_x \\
& \mathsf{x}\{\vec{t}\}[\vec{v}] := \mathsf{x}\{\vec{t}[\vec{v}]\} \\
& f(\mathbf{t})[\vec{v}] := f(\mathbf{t}[\vec{v}])
\end{aligned}
$$

$$
\begin{aligned}
& - [-] : \mathsf{Tm}\ \theta_2\ \gamma \longrightarrow \mathsf{MSub}\ \theta_1\ \delta\ \theta_2 \longrightarrow \mathsf{Tm}\ \theta_1\ \delta.\gamma \\
& x[\mathbf{v}] := x \\
& \mathsf{x}\{\vec{t}\}[\mathbf{v}] := \mathbf{v}_{\mathsf{x}}[\mathsf{id}, \vec{t}[\mathbf{v}]] \\
& f(\mathbf{t})[\mathbf{v}] := f(\mathbf{t}[\mathbf{v}])
\end{aligned}
$$

$$
\begin{aligned}
& - [-] : \mathsf{Sub}\ \theta\ \gamma_2\ \delta \longrightarrow \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2 \longrightarrow \mathsf{Sub}\ \theta\ \gamma_1\ \delta \\
& \varepsilon[\vec{v}] := \varepsilon \\
& (\vec{t}, u)[\vec{v}] := \vec{t}[\vec{v}], u[\vec{v}]
\end{aligned}
$$

$$
\begin{aligned}
& - [-] : \mathsf{Sub}\ \theta_2\ \gamma\ \gamma_0 \longrightarrow \mathsf{MSub}\ \theta_1\ \delta\ \theta_2 \longrightarrow \mathsf{Sub}\ \theta_1\ \delta.\gamma\ \gamma_0 \\
& \varepsilon[\mathbf{v}] := \varepsilon \\
& (\vec{t}, u)[\mathbf{v}] := \vec{t}[\mathbf{v}], u[\mathbf{v}]
\end{aligned}
$$

$$
\begin{aligned}
& - [-] : \mathsf{MSub}\ \theta\ \gamma_2\ \xi \longrightarrow \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2 \longrightarrow \mathsf{MSub}\ \theta\ \gamma_1\ \xi \\
& \varepsilon[\vec{v}] := \varepsilon \\
& (\mathbf{t}, \vec{x}_\delta.u)[\vec{v}] := \mathbf{t}[\vec{v}], \vec{x}.u[\vec{v}, \mathsf{id}]
\end{aligned}
$$

$$
\begin{aligned}
& - [-] : \mathsf{MSub}\ \theta_2\ \gamma\ \xi \longrightarrow \mathsf{MSub}\ \theta_1\ \delta\ \theta_2 \longrightarrow \mathsf{MSub}\ \theta_1\ \delta.\gamma\ \xi \\
& \varepsilon[\mathbf{v}] := \varepsilon \\
& (\mathbf{t}, \vec{x}.u)[\mathbf{v}] := \mathbf{t}[\mathbf{v}], \vec{x}.u[\mathbf{v}]
\end{aligned}
$$

$$
\begin{aligned}
& - [-] : \mathsf{Ctx}\ \theta\ \gamma_2 \longrightarrow \mathsf{Sub}\ \theta\ \gamma_1\ \gamma_2 \longrightarrow \mathsf{Ctx}\ \theta\ \gamma_1 \\
& (\cdot)[\vec{v}] := \cdot \\
& (\Gamma, x : T)[\vec{v}] := \Gamma[\vec{v}], x : T[\vec{v}, \mathsf{id}]
\end{aligned}
$$

$$
\begin{aligned}
& - [-] : \mathsf{Ctx}\ \theta_2\ \gamma \longrightarrow \mathsf{MSub}\ \theta_1\ \delta\ \theta_2 \longrightarrow \mathsf{Ctx}\ \theta_1\ \delta.\gamma \\
& (\cdot)[\mathbf{v}] := \cdot \\
& (\Gamma, x : T)[\mathbf{v}] := \Gamma[\mathbf{v}], x : T[\mathbf{v}]
\end{aligned}
$$

Figure 7.1: Application of a variable or metavariable substitution

Example 7.2. If $t \in \mathsf{Tm}\ (\cdot)\ (\gamma, x)$ and $u \in \mathsf{Tm}\ (\cdot)\ \gamma$, then by applying the metavariable substitution $x.t, u \in \mathsf{MSub}\ (\cdot)\ \gamma\ (\mathsf{t}\{x\}, \mathsf{u})$ to the term $@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \in \mathsf{Tm}\ (\mathsf{t}\{x\}, \mathsf{u})\ (\cdot)$ we get the term $@(\lambda(x.t), u) \in \mathsf{Tm}\ (\cdot)\ \gamma$. □

Remark 7.4. Compared to frameworks derived from contextual modal type theory [NPP08], our metavariable substitutions are not required to be closed and can introduce new variables in the scope of the resulting term. For instance, in the previous example, while the term $@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u})$ lives in an empty variable scope, the application of the metavariable substitution yields $@(\lambda(x.t), u)$, which lives in the scope γ. Therefore, a metavariable $\mathsf{t}\{x_1, \ldots, x_k\}$ should not be seen as a placeholder for a term containing only $x_1, \ldots, x_k$, but instead for a term in *any* scope extended by $x_1, \ldots, x_k$. □

Substitution application satisfies the following basic laws.

Proposition 7.1 (Unit laws for id). *We have* $e[\mathsf{id}_\gamma] = e$ *and* $e[\mathsf{id}_\theta] = e$ *for all* $e \in \mathsf{Expr}\ \theta\ \gamma$, *and* $\mathsf{id}_\delta[\vec{t}] = \vec{t}$ *for all* $\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta$, *and* $\mathsf{id}_\theta[\mathbf{v}] = \mathbf{v}$ *for all* $\mathbf{v} \in \mathsf{MSub}\ \xi\ \gamma\ \theta$.

Proof. We first show $e[\mathsf{id}_\gamma] = e$ by induction on e and $\mathsf{id}_\delta[\vec{t}] = \vec{t}$ by induction on δ. We then show $e[\mathsf{id}_\theta] = e$ by induction on e and $\mathsf{id}_\theta[\mathbf{v}] = \mathbf{v}$ by induction on θ. ■

The following two properties are shown simultaneously.

Proposition 7.2 (Commutation lemmas). *We have* $e[\mathbf{u}][\vec{v}, \mathrm{id}_\delta] = e[\mathbf{u}[\vec{v}]] \in \mathsf{Expr}\ \theta_1\ \gamma_1.\delta$ *for all* $e \in \mathsf{Expr}\ \theta_2\ \delta$ *and* $\mathbf{u} \in \mathsf{MSub}\ \theta_1\ \gamma_2\ \theta_2$ *and* $\vec{v} \in \mathsf{Sub}\ \theta_1\ \gamma_1\ \gamma_2$*. We have* $e[\vec{u}][\mathbf{v}] = e[\mathbf{v}][\mathrm{id}_\delta, \vec{u}[\mathbf{v}]] \in \mathsf{Expr}\ \theta_1\ \delta.\gamma_1$ *for all* $e \in \mathsf{Expr}\ \theta_2\ \gamma_2$ *and* $\vec{u} \in \mathsf{Sub}\ \theta_2\ \gamma_1\ \gamma_2$ *and* $\mathbf{v} \in \mathsf{MSub}\ \theta_1\ \delta\ \theta_2$*.*

Proposition 7.3 (Associativity of substitution). *Let* $e \in \mathsf{Expr}\ \theta_3\ \gamma_3$*. For all* $\vec{v} \in \mathsf{Sub}\ \theta_3\ \gamma_2\ \gamma_3$ *and* $\vec{u} \in \mathsf{Sub}\ \theta_3\ \gamma_1\ \gamma_2$ *we have* $e[\vec{v}][\vec{u}] = e[\vec{v}[\vec{u}]]$*, and for all* $\mathbf{v} \in \mathsf{MSub}\ \theta_2\ \gamma_3\ \theta_3$ *and* $\mathbf{u} \in \mathsf{MSub}\ \theta_1\ \gamma_3\ \theta_2$ *we have* $e[\mathbf{v}][\mathbf{u}] = e[\mathbf{v}[\mathbf{u}]]$*.*

Proof. By induction on e, on the following order: first $e[\vec{v}][\vec{u}] = e[\vec{v}[\vec{u}]]$, then $e[\mathbf{u}][\vec{v}, \mathrm{id}_\delta] = e[\mathbf{u}[\vec{v}]]$, then $e[\vec{u}][\mathbf{v}] = e[\mathbf{v}][\mathrm{id}_\delta, \vec{u}[\mathbf{v}]]$, then $e[\mathbf{v}][\mathbf{u}] = e[\mathbf{v}[\mathbf{u}]]$. ■

Patterns

Given an underlying signature Σ, *term patterns* and *metavariables substitution patterns* are defined by the following grammars.

$$\begin{array}{rrll}
\boxed{\mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma} \ni & t, u, v, s ::= & \mid \mathsf{x}\{\mathrm{id}_\gamma\} & \text{if } \theta = \mathsf{x}\{\gamma\} \\
 & & \mid f(\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi) & \text{if } f(\xi) \in \Sigma \\
\boxed{\mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi} \ni & \mathbf{t}, \mathbf{u}, \mathbf{s}, \mathbf{v} ::= & \mid \varepsilon & \text{if } \xi = \cdot \text{ and } \theta = \cdot \\
 & & \mid \mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta_1\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{Tm}^{\mathsf{P}}\ \theta_2\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\} \text{ and } \theta = \theta_1.\theta_2
\end{array}$$

Compared with the regular syntax, the pattern condition imposes that each metavariable $\mathsf{x}\{\delta\} \in \theta$ must occur precisely once, and moreover applied to all variables occurring in the scope γ of its occurrence — in particular, imposing δ to be equal to γ. These restrictions ensure that, differently from regular terms, patterns support decidable and unitary matching [Mil91].

Example 7.3. In $\Sigma_{\lambda\Pi}$, we can build the pattern $@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \in \mathsf{Tm}^{\mathsf{P}}\ (\mathsf{t}\{x\}, \mathsf{u})\ (\cdot)$. □

Note that we have inclusions $\mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma \subset \mathsf{Tm}\ \theta\ \gamma$ and $\mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi \subset \mathsf{MSub}\ \theta\ \gamma\ \xi$, which we use to implicitly coerce patterns into regular expressions when needed.

Rewriting

Given an underlying signature Σ, the notion of pattern then allow us to define *rewrite rules*, which are of the form

$$\theta \Vdash l \in \mathsf{Tm}^{\mathsf{P}}\ \theta\ (\cdot) \longmapsto r \in \mathsf{Tm}\ \theta\ (\cdot)$$

where l is not a metavariable. Note that, because our patterns are linear, then our rewrite rules are automatically left-linear.

$$\boxed{t \longrightarrow u \quad (t \in \mathsf{Tm}\ \theta\ \gamma;\ u \in \mathsf{Tm}\ \theta\ \gamma)} \qquad \boxed{\Gamma \longrightarrow \Delta \quad (\Gamma \in \mathsf{Ctx}\ \theta\ \gamma;\ \Delta \in \mathsf{Ctx}\ \theta\ \gamma)}$$

$$\frac{\vec{t} \longrightarrow \vec{t}'}{\mathsf{x}\{\vec{t}\} \longrightarrow \mathsf{x}\{\vec{t}'\}} \qquad \frac{\mathbf{v} \longrightarrow \mathbf{v}'}{f(\mathbf{v}) \longrightarrow f(\mathbf{v}')} \qquad \frac{\Gamma \longrightarrow \Gamma'}{\Gamma, x : T \longrightarrow \Gamma', x : T}$$

$$\frac{(\theta_0 \Vdash l \longmapsto r) \in \mathcal{R} \qquad \mathbf{t} \in \mathsf{MSub}\ \theta\ \gamma\ \theta_0}{l[\mathbf{t}] \longrightarrow r[\mathbf{t}]} \qquad \frac{T \longrightarrow T'}{\Gamma, x : T \longrightarrow \Gamma, x : T'}$$

$$\boxed{\vec{t} \longrightarrow \vec{u} \quad (\vec{t} \in \mathsf{Sub}\ \theta\ \gamma\ \delta;\ \vec{u} \in \mathsf{Sub}\ \theta\ \gamma\ \delta)} \qquad \boxed{\mathbf{v} \longrightarrow \mathbf{u} \quad (\mathbf{v} \in \mathsf{MSub}\ \theta\ \gamma\ \xi;\ \mathbf{u} \in \mathsf{MSub}\ \theta\ \gamma\ \xi)}$$

$$\frac{\vec{t} \longrightarrow \vec{t}'}{\vec{t}, u \longrightarrow \vec{t}', u} \qquad \frac{u \longrightarrow u'}{\vec{t}, u \longrightarrow \vec{t}, u'} \qquad \frac{\mathbf{v} \longrightarrow \mathbf{v}'}{\mathbf{v}, \vec{x}.u \longrightarrow \mathbf{v}', \vec{x}.u} \qquad \frac{u \longrightarrow u'}{\mathbf{v}, \vec{x}.u \longrightarrow \mathbf{t}, \vec{x}.u'}$$

Figure 7.2: Rewriting relation defined by rewrite system $\mathcal{R}$

Example 7.4. In the signature $\Sigma_{\lambda\Pi}$, we can define the β-rule as

$$@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$$

where we have omitted the metavariable scope, as it can be straightforwardly reconstructed by inspecting the left-hand side. □

A *rewrite system* (over Σ) is then simply a set of rewrite rules (over Σ), and given a rewrite system $\mathcal{R}$ we define the rewriting relation $e \longrightarrow e'$ in Figure 7.2. In order for it to be well-defined, this definition must be done simultaneously with a proof that reduction preserves underlying scopes: we have $|\Gamma| = |\Gamma'|$ whenever $\Gamma \longrightarrow \Gamma'$, by an easy induction on Γ. The relations $\longrightarrow^*$ and $\equiv$ are then defined as usual, respectively as the reflexive-transitive and reflexive-symmetric-transitive closures of $\longrightarrow$. The relation $\equiv$ is called *definitional equality* (or *conversion*).

One of the key properties of rewriting is its stability under substitution:

Proposition 7.4 (Stability of rewriting under substitution). *Let $e \in \mathsf{Expr}\ \theta\ \gamma$ with $e \longrightarrow^* e'$. If $\vec{v} \in \mathsf{Sub}\ \theta\ \delta\ \gamma$ and $\vec{v} \longrightarrow^* \vec{v}'$ then $e[\vec{v}] \longrightarrow^* e'[\vec{v}']$. If $\mathbf{v} \in \mathsf{MSub}\ \xi\ \gamma\ \theta$ and $\mathbf{v} \longrightarrow^* \mathbf{v}'$ then $e[\mathbf{v}] \longrightarrow^* e'[\mathbf{v}']$.*

Proof. We first show that $\vec{v} \longrightarrow^* \vec{v}'$ implies $e[\vec{v}] \longrightarrow^* e[\vec{v}']$, by induction on e, and then that $e \longrightarrow e'$ implies $e[\vec{v}] \longrightarrow e'[\vec{v}]$, by induction on $e \longrightarrow e'$. By iterating these two statements, we then conclude that $e \longrightarrow^* e'$ and $\vec{v} \longrightarrow^* \vec{v}'$ imply $e[\vec{v}] \longrightarrow^* e'[\vec{v}']$.

Then, we show that $\mathbf{v} \longrightarrow^* \mathbf{v}'$ implies $e[\mathbf{v}] \longrightarrow^* e[\mathbf{v}']$, by induction on e, and then that $e \longrightarrow e'$ implies $e[\mathbf{v}] \longrightarrow e'[\mathbf{v}]$, by induction on $e \longrightarrow e'$. By iterating these two statements, we then conclude that $e \longrightarrow^* e'$ and $\mathbf{v} \longrightarrow^* \mathbf{v}'$ imply $e[\mathbf{v}] \longrightarrow^* e'[\mathbf{v}']$. ■

$$\begin{array}{rll}
\boxed{\mathsf{Thy}} \ni \mathbb{T} ::= & \mid \cdot & \\
& \mid \mathbb{T},\ f(\Xi \in \mathsf{MCtx}_{|\mathbb{T}|})\ \mathsf{sort} & \\
& \mid \mathbb{T},\ f(\Xi_1 \in \mathsf{MCtx}_{|\mathbb{T}|}\ ;\ \Xi_2 \in \mathsf{MCtx}_{|\mathbb{T}|}\ |\Xi_1|\ ; & \\
& \qquad \vec{t} \in \mathsf{Tm}_{|\mathbb{T}|}\ |\Xi_1.\Xi_2|\ \gamma \equiv \vec{u} \in \mathsf{Tm}_{|\mathbb{T}|}\ |\Xi_1.\Xi_2|\ \gamma) : T \in \mathsf{Tm}_{|\mathbb{T}|}\ |\Xi_1.\Xi_2|\ (\cdot) & \\
& \mid \mathbb{T},\ \theta \Vdash l \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ \theta\ (\cdot) \longmapsto r \in \mathsf{Tm}_{|\mathbb{T}|}\ \theta\ (\cdot) & \text{with } l \text{ not a metavariable}
\end{array}$$

$$\begin{array}{ll}
|-| : \mathsf{Thy} \to \mathsf{Sig} & \\
|\cdot| := \cdot & |\mathbb{T}, f(\Xi_1\ ;\ \Xi_2\ ;\ \vec{u} \equiv \vec{t}) : U| := |\mathbb{T}|, f(|\Xi_2|) \\
|\mathbb{T}, f(\Xi)\ \mathsf{sort}| := |\mathbb{T}|, f(|\Xi|) & |\mathbb{T}, \theta \Vdash l \longmapsto r| := |\mathbb{T}|
\end{array}$$

Figure 7.3: Definition of theories

This implies in particular that conversion is stable under substitution.

Corollary 7.1 (Stability of conversion under substitution). *Suppose $e \equiv e'$. We have $e[\vec{v}] \equiv e'[\vec{v}']$ for all $\vec{v} \equiv \vec{v}'$ and $e[\mathbf{v}] \equiv e'[\mathbf{v}']$ for all $\mathbf{v} \equiv \mathbf{v}'$.*

Remark 7.5. Our notion of rewriting corresponds roughly to Hamana's Second-Order Computation Systems [Ham22], which are Second-Order Algebraic Theories (SOATs) whose equations are all rewrite rules. Our rewrite systems can also be seen either as a second-order restriction of Nipkow's Higher-Order Rewrite Systems (HRSs) [MN98], or as a simply typed version of Klop's Combinatory Reduction Systems (CRSs) [Kvv93] over a single base type. Importantly, this allows us to use in our framework confluence criteria developed in these settings. □

7.2 Theories

We now come to a central definition of this chapter, that of a *theory* $\mathbb{T}$, defined in Figure 7.3. Note that, similarly to contexts and metavariable contexts, they are defined simultaneously with a function $|-|$, this time computing the *underlying signature* $|\mathbb{T}| \in \mathsf{Sig}$ of a theory $\mathbb{T}$. Apart from rewrite rules, theories are made of *schematic typing rules*, which can be either *sort rules* or *term rules*. Let us now explain them in detail.

Sort rules

In our framework, a *sort* T is a term that can appear in the second position of the typing judgment $t : T$, and they are used to represent the *judgment forms* of the theory.[5] For instance, vanilla Martin-Löf Type Theory features two judgment forms: A type, for asserting that A is a type, and $t : A$ for asserting that t is a term of type A. In our framework, these are materialized by the following sort rules.

$$\frac{}{\mathrm{Ty}\ \mathsf{sort}} \qquad\qquad \frac{\mathtt{A} : \mathrm{Ty}}{\mathrm{Tm}(\mathtt{A})\ \mathsf{sort}}$$

Formally, a sort rule is of the form

$$c(\Xi \in \mathsf{MCtx})\ \mathsf{sort}$$

and the previously shown rules are just an informal notation for $\mathrm{Ty}(\cdot)$ sort and $\mathrm{Tm}(\mathtt{A} : \mathrm{Ty})$ sort. More precisely, our notation represents metavariables $\mathtt{x}\{\Gamma\} : T \in \Xi$ as premises $\Gamma \vdash \mathtt{x} : T$, or just $\mathtt{x} : T$ when Γ is empty. In the following, we will make use of such informal representations in order to enhance readability of schematic rules.

Term rules

As motivated in Chapter 6, we would like in our framework to allow for specifying theories in which some of the arguments are omitted. In order to do this, we separate the arguments of a term rule into a metavariable context Ξ_1 of erased arguments and a metavariable context Ξ_2 of arguments present in the syntax. Moreover, for reasons that will be clear in Chapter 8, we also consider *equational hypotheses* $\vec{t} \equiv \vec{u}$ in our rules, which need to be verified when applying the rule, leading to term rules of the form

$$f(\Xi_1 \in \mathsf{MCtx}\ ;\ \Xi_2 \in \mathsf{MCtx}\ |\Xi_1|\ ;\ \vec{t} \in \mathsf{Tm}\ |\Xi_1.\Xi_2|\ \gamma \equiv \vec{u} \in \mathsf{Tm}\ |\Xi_1.\Xi_2|\ \gamma) : T \in \mathsf{Tm}\ |\Xi_1.\Xi_2|\ (\cdot)$$

Note that, whereas Ξ_1 is closed, Ξ_2 is allowed to depend on the underlying scope of Ξ_1, so the sorts and contexts of the non-omitted arguments can depend on the missing information, and $\vec{t}$, $\vec{u}$ and T are allowed to depend on the underlying scopes of both Ξ_1 and Ξ_2. As expected, only the arguments from Ξ_2 are kept when calculating the underlying signature in Figure 7.3.

Three examples of rules fitting this definition are the following ones for Π, λ and @ — note however that the one for Π is slightly degenerate, given that we have $\Xi_1 = \cdot$ and

[5] We depart from the Dedukti terminology by purposely avoiding calling them "types", in order to prevent a name clash with the types of the theories we define. Still, we allow ourselves to say "t is typed by sort T" to mean $t : T$.

thus no erased premises.

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty}}{\Pi(\mathsf{A}, x.\mathsf{B}\{x\}) : \mathrm{Ty}} \qquad\qquad \frac{\begin{array}{c}\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})\end{array}}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t}, \mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})}$$

Once again, we have presented the rules using the informal notation, which can be parsed into the formal one in the following manner:

$$\begin{array}{l}
\Pi(\cdot\ ;\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\varepsilon \equiv \varepsilon) : \mathrm{Ty} \\
\lambda(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})\ ;\varepsilon \equiv \varepsilon) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \\
@(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})), \mathsf{u} : \mathrm{Tm}(\mathsf{A})\ ;\varepsilon \equiv \varepsilon) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})
\end{array}$$

Finally, to represent equation hypotheses $\vec{t} \equiv \vec{u}$ with our informal notation, we let $\vec{t} = t_1, \dots, t_k$ and $\vec{u} = u_1, \dots, u_k$ and write a premise $t_i \equiv u_i$ for each i. For instance, we can define the rule for the constructor refl for the equality type as[6]

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad \mathsf{a} : \mathrm{Tm}(\mathsf{A}) \qquad \mathsf{b} : \mathrm{Tm}(\mathsf{A}) \qquad \mathsf{a} \equiv \mathsf{b}}{\mathrm{refl} : \mathrm{Tm}(\mathrm{Eq}(\mathsf{A}, \mathsf{a}, \mathsf{b}))}$$

which can be parsed into the formal notation as

$$\mathrm{refl}(\mathsf{A} : \mathrm{Ty},\ \mathsf{a} : \mathrm{Tm}(\mathsf{A}),\ \mathsf{b} : \mathrm{Tm}(\mathsf{A})\ ;\cdot\ ;\mathsf{a} \equiv \mathsf{b}) : \mathrm{Tm}(\mathrm{Eq}(\mathsf{A}, \mathsf{a}, \mathsf{b}))$$

Example 7.5. By putting together some of the rules seen in this section, we get the following theory $\mathbb{T}_{\lambda\Pi}$ defining a basic version of MLTT with only dependent functions.

$$\begin{array}{l}
\mathrm{Ty}(\cdot)\ \text{sort},\quad \mathrm{Tm}(\mathsf{A} : \mathrm{Ty})\ \text{sort},\quad \Pi(\cdot\ ;\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\varepsilon \equiv \varepsilon) : \mathrm{Ty}, \\
\lambda(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})\ ;\varepsilon \equiv \varepsilon) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})), \\
@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\ \mathsf{u} : \mathrm{Tm}(\mathsf{A})\ ;\varepsilon \equiv \varepsilon) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}), \\
@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}
\end{array} \qquad (\mathbb{T}_{\lambda\Pi})$$

When computing its underlying signature $|\mathbb{T}_{\lambda\Pi}|$ we get the signature $\Sigma_{\lambda\Pi}$. □

7.3 Typing rules

In the previous section, we have seen that a theory $\mathbb{T}$ is specified by rewrite rules and schematic typing rules. These schematic rules can be instantiated into concrete typing

[6] Of course, it would be possible to eliminate the equational hypothesis and the argument b, and replace the sort by $\mathrm{Tm}(\mathrm{Eq}(\mathsf{A}, \mathsf{a}, \mathsf{a}))$, which would yield an equivalent rule. However, for the purposes of bidirectional typing, we need the sort of this rule to be a pattern and hence linear, as we explain in Section 8.1.

$$\boxed{\Theta \vdash \quad (\Theta \in \mathsf{MCtx})} \qquad \boxed{\Theta;\Gamma \vdash \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|)}$$

$$\textsc{EmptyMCtx}\ \frac{}{\cdot \vdash} \qquad \textsc{ExtMCtx}\ \frac{\Theta;\Gamma \vdash T\ \mathsf{sort}}{\Theta, \mathsf{x}\{\Gamma\} : T \vdash} \qquad \textsc{EmptyCtx}\ \frac{\Theta \vdash}{\Theta;\cdot \vdash} \qquad \textsc{ExtCtx}\ \frac{\Theta;\Gamma \vdash T\ \mathsf{sort}}{\Theta;\Gamma, x : T \vdash}$$

$$\boxed{\Theta;\Gamma \vdash T\ \mathsf{sort} \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ T \in \mathsf{Tm}\ |\Theta|\ |\Gamma|)}$$

$$\textsc{SortSym}\quad f(\Xi)\ \mathsf{sort} \in \mathbb{T}\ \frac{\Theta;\Gamma \vdash \mathbf{t} : \Xi}{\Theta;\Gamma \vdash f(\mathbf{t})\ \mathsf{sort}}$$

$$\boxed{\Theta;\Gamma \vdash t : T \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ T \in \mathsf{Tm}\ |\Theta|\ |\Gamma|;\ t \in \mathsf{Tm}\ |\Theta|\ |\Gamma|)}$$

$$\textsc{Var}\quad x : T \in \Gamma\ \frac{\Theta;\Gamma \vdash}{\Theta;\Gamma \vdash x : T} \qquad \textsc{MVar}\quad \mathsf{x}\{\Delta\} : T \in \Theta\ \frac{\Theta;\Gamma \vdash \vec{t} : \Delta}{\Theta;\Gamma \vdash \mathsf{x}\{\vec{t}\} : T[\vec{t}]}$$

$$\textsc{TmSym}\quad \begin{array}{c} f(\Xi_1 \mathbin{\boldsymbol{;}} \Xi_2 \mathbin{\boldsymbol{;}} \vec{t} \equiv \vec{u}) : T \in \mathbb{T} \\ \vec{t}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1, \mathbf{t}_2] \end{array}\ \frac{\Theta;\Gamma \vdash \mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2 \qquad \Theta;\Gamma \vdash T[\mathbf{t}_1, \mathbf{t}_2]\ \mathsf{sort}}{\Theta;\Gamma \vdash f(\mathbf{t}_2) : T[\mathbf{t}_1, \mathbf{t}_2]}$$

$$\textsc{Conv}\quad T \equiv U\ \frac{\Theta;\Gamma \vdash t : T \qquad \Theta;\Gamma \vdash U\ \mathsf{sort}}{\Theta;\Gamma \vdash t : U}$$

$$\boxed{\Theta;\Gamma \vdash \vec{t} : \Delta \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ \Delta \in \mathsf{Ctx}\ |\Theta|;\ \vec{t} \in \mathsf{Sub}\ |\Theta|\ |\Gamma|\ |\Delta|)}$$

$$\textsc{EmptySub}\ \frac{\Theta;\Gamma \vdash}{\Theta;\Gamma \vdash \varepsilon : (\cdot)} \qquad \textsc{ExtSub}\ \frac{\Theta;\Gamma \vdash \vec{t} : \Delta \qquad \Theta;\Gamma \vdash t : T[\vec{t}]}{\Theta;\Gamma \vdash \vec{t}, t : (\Delta, x : T)}$$

$$\boxed{\Theta;\Gamma \vdash \mathbf{t} : \Xi \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx}\ |\Theta|;\ \Xi \in \mathsf{MCtx};\ \mathbf{t} \in \mathsf{MSub}\ |\Theta|\ |\Gamma|\ |\Xi|)}$$

$$\textsc{EmptyMSub}\ \frac{\Theta;\Gamma \vdash}{\Theta;\Gamma \vdash \varepsilon : (\cdot)} \qquad \textsc{ExtMSub}\ \frac{\Theta;\Gamma \vdash \mathbf{t} : \Xi \qquad \Theta;\Gamma.\Delta[\mathbf{t}] \vdash t : T[\mathbf{t}]}{\Theta;\Gamma \vdash \mathbf{t}, \vec{x}_{\Delta}.t : (\Xi, \mathsf{x}\{\Delta\} : T)}$$

Figure 7.4: Typing system defined by the theory $\mathbb{T}$

rules, defining the type system of the corresponding theory, with the rules in Figure 7.4. There, we write $\equiv$ for the conversion generated by the underlying rewrite system of $\mathbb{T}$.

The type system is defined by 6 judgment forms:

- $\Theta \vdash$: Well-typedness of metavariable context Θ.
- $\Theta; \Gamma \vdash$: Well-typedness of variable context Γ under metavariable context Θ.
- $\Theta; \Gamma \vdash T$ sort : Well-typedness of sort T under contexts $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash t : T$: Typing of a term t by T under context $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash \vec{t} : \Delta$: Typing of a variable substitution $\vec{t}$ by Δ under context $\Theta; \Gamma$.
- $\Theta; \Gamma \vdash \mathbf{t} : \Xi$: Typing of a metavariable substitution $\mathbf{t}$ by Ξ under context $\Theta; \Gamma$.

The most important rules are the ones which instantiate schematic typing rules, which are SortSym and TmSym. For instance, in order to use TmSym to type $f(\mathbf{t}_2)$ a metavariable substitution $\mathbf{t}_1$ not stored in the syntax must be "guessed", and then we must show that $\mathbf{t}_1, \mathbf{t}_2$ is typed by $\Xi_1.\Xi_2$, verify the equational premises of the rule and check that the sort of the rule is well-typed. In order to type $\mathbf{t}_1, \mathbf{t}_2$, we can apply the rules for typing metavariable substitutions, which has the effect of unfolding the judgment $\mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2$ into regular term typing judgments. At the end of this unfolding process, the resulting "big-step derivation" has basically the same shape as the schematic typing rule for f, and it can be understood as its instantiation. Let us look at a concrete example of this.

Example 7.6. Suppose we want to show that $@(t, u)$ is well-typed in the theory $\mathbb{T}_{\lambda\Pi}$. Because $@$ has the term rule

$$@(\mathsf{A} : \mathrm{Ty}, \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})), \mathsf{u} : \mathrm{Tm}(\mathsf{A})\ ;\ \varepsilon \equiv \varepsilon) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})$$

then by guessing some A and B we can start the derivation with rule TmSym, giving

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash A, x.B, t, u : (\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty},\ \mathsf{t} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\ \mathsf{u} : \mathrm{Tm}(\mathsf{A}))\\ \Theta; \Gamma \vdash \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u] \text{ sort}\end{array}}{\Theta; \Gamma \vdash @(t, u) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u]}$$

If we note that $\mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u] = \mathrm{Tm}(B[\mathrm{id}_\Gamma, u])$, and we continue by applying the rules defining the judgment $\Theta; \Gamma \vdash \mathbf{t} : \Xi$, we get

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash \qquad \Theta; \Gamma \vdash A : \mathrm{Ty} \qquad \Theta; \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty}\\ \Theta; \Gamma \vdash t : \mathrm{Tm}(\Pi(A, x.B)) \qquad \Theta; \Gamma \vdash u : \mathrm{Tm}(A) \qquad \Theta; \Gamma \vdash \mathrm{Tm}(B[\mathrm{id}_\Gamma, u]) \text{ sort}\end{array}}{\Theta; \Gamma \vdash @(t, u) : \mathrm{Tm}(B[\mathrm{id}_\Gamma, u])}$$

which can be understood as the instantiation of the schematic rule for $@$. Note that sometimes the first three and the last premises are omitted, but this is only justified because they are admissible from the other ones, a result we could also show here by applying results from Section 7.5. □

$$\boxed{\mathbb{T} \vdash \quad (\mathbb{T} \in \mathsf{Thy})}$$

$$\frac{}{\cdot \vdash} \qquad \frac{\mathbb{T} \vdash \qquad \mathbb{T} \triangleright \Xi \vdash}{\mathbb{T}, f(\Xi)\ \mathsf{sort} \vdash} \qquad \frac{\mathbb{T} \vdash}{\mathbb{T}, (\theta \Vdash l \longmapsto r) \vdash}$$

$$\frac{\mathbb{T} \vdash \qquad \mathbb{T} \triangleright \Xi_1.\Xi_2 \vdash T\ \mathsf{sort} \qquad \text{for some } \Delta : \quad \mathbb{T} \triangleright \Xi_1.\Xi_2 \vdash \vec{t} : \Delta \qquad \mathbb{T} \triangleright \Xi_1.\Xi_2 \vdash \vec{u} : \Delta}{\mathbb{T}, f(\Xi_1 \ ; \Xi_2 \ ; \vec{t} \equiv \vec{u}) : T \vdash}$$

Figure 7.5: Well-typed theories

Remark 7.6. In rule TmSym it might seem odd that we also ask the sort to be well-typed, whereas this hypothesis is not needed in rules MVar and Var. The reason is that in the proof of Theorem 8.4 we will need to apply the induction hypothesis to the sort T of the term $f(\mathbf{t})$, and thus we need a derivation of T sort smaller than the one of $f(\mathbf{t}) : T$ we start with. Nevertheless, we will show in Section 7.5 that this extra hypothesis is admissible, allowing us to use the economic version of the rule when building derivations. A similar technique is also employed by Harper and Pfenning [HP05], and Abel *et al.* [AOV18]. □

Notation 7.3. We finish this subsection by establishing some notations.

1. We write $\Theta; \Gamma \vdash \mathcal{J}$ for any of the following: $\Theta; \Gamma \vdash$ or $\Theta; \Gamma \vdash T$ sort or $\Theta; \Gamma \vdash t : T$ or $\Theta; \Gamma \vdash \vec{t} : \Delta$ or $\Theta; \Gamma \vdash \mathbf{t} : \Xi$.
2. We write $\mathbb{T} \triangleright \Theta; \Gamma \vdash \mathcal{J}$ when $\mathbb{T}$ is not clear from the context.
3. We write $\Theta \vdash \mathcal{J}$ for $\Theta; \cdot \vdash \mathcal{J}$ and $\Gamma \vdash \mathcal{J}$ for $\cdot; \Gamma \vdash \mathcal{J}$. □

7.4 Valid theories

Our definition of theories given in Section 7.2 specifies the desired syntax but imposes not typing constraints whatsoever, allowing for non-sensible and ill-behaved theories. But now that we have introduced typing rules, we can impose such constraints *a posteriori* by defining the *valid theories*.

Our first step to do this is to define the *well-typed theories* with the judgment $\mathbb{T} \vdash$ specified in Figure 7.5. The definition of $\mathbb{T} \vdash$ ensures that each time we extend a theory $\mathbb{T}$ with a schematic typing, $\mathbb{T}$ can justify that the new rule is well-typed. For sort rules $f(\Xi)$ sort this amounts to ensuring that the metavariable context Ξ is well-typed, whereas for term rules $f(\Xi_1 \ ; \Xi_2 \ ; \vec{t} \equiv \vec{u}) : T$ this means ensuring that T is a well-typed sort in metavariable context $\Xi_1.\Xi_2$ — implying in particular that the metavariable context is also well-typed — and that there is some context Δ typing both $\vec{t}$ and $\vec{u}$ under the metavariable context $\Xi_1.\Xi_2$.

The definition of well-typed theory ensures that the schematic rules are well-behaved, but it does not say anything about the rewrite rules of the theory. To remedy this, we then define a theory $\mathbb{T}$ to be *valid* when the following criteria are met:

- **(I)** $\mathbb{T}$ is well-typed.
- **(II)** The rewrite system of $\mathbb{T}$ is *confluent*, meaning that $t_2 \ {}^*\!\longleftarrow\ t_1 \longrightarrow^* t_3$ implies $t_2 \longrightarrow^* t_4 \ {}^*\!\longleftarrow\ t_3$ for some t_4.
- **(III)** The rewrite system of $\mathbb{T}$ satisfies *subject reduction*, meaning that $\mathbb{T} \triangleright \Theta; \Gamma \vdash t : T$ and $t \longrightarrow t'$ imply $\mathbb{T} \triangleright \Theta; \Gamma \vdash t' : T$, and that $\mathbb{T} \triangleright \Theta; \Gamma \vdash T$ sort and $T \longrightarrow T'$ imply $\mathbb{T} \triangleright \Theta; \Gamma \vdash T'$ sort.[7]

Example 7.7. It is tedious but uncomplicated to see that the theory $\mathbb{T}_{\lambda\Pi}$ is valid. Checking its well-typedness is straightforward, and confluence follows from the fact that the rewrite system is orthogonal [MN98]. The most interesting part is verifying that the rewrite system satisfies subject reduction, which we postpone to Example 7.8. □

Remark 7.7. The definition of $\mathbb{T} \vdash$ asks schematic typing rules to be typed incrementally, which excludes theories that rely on circularities — for instance, when a rule depends on itself to be well-typed. Nevertheless, by a form of weakening for theories we can still deduce that all schematic typing rules of $\mathbb{T}$ can also be typed in $\mathbb{T}$ itself (for instance, if $f(\Xi)$ sort $\in \mathbb{T}$ and $\mathbb{T} \vdash$ then $\mathbb{T} \triangleright \Xi \vdash$), a fact that we will often use without announcement in the proofs to come. □

7.5 Metatheory

We now show some basic metaproperties satisfied by the declarative type system. Most of these properties hold even when the theory is not well-typed or valid, so such assumptions will be stated explicitly when needed.

Proposition 7.5 (Weakening). *Let us write $\Gamma \sqsubseteq \Delta$ if Γ is a subsequence of Δ, and $\Theta \sqsubseteq \Xi$ if Θ is a subsequence of Ξ. The following rules are admissible.*

$$\Gamma \sqsubseteq \Delta\,\frac{\Theta; \Gamma \vdash \mathcal{J} \qquad \Theta; \Delta \vdash}{\Theta; \Delta \vdash \mathcal{J}} \qquad\qquad \Theta \sqsubseteq \Xi\,\frac{\Theta; \Gamma \vdash \mathcal{J} \qquad \Xi \vdash}{\Xi; \Gamma \vdash \mathcal{J}}$$

Proof. In order for the induction to go through, we strengthen the first statement: instead we show that $\Theta; \Gamma.\Gamma' \vdash \mathcal{J}$ and $\Theta; \Delta \vdash$ and $\Gamma \sqsubseteq \Delta$ imply $\Theta; \Delta.\Gamma' \vdash \mathcal{J}$. The proof is then by induction on $\Theta; \Gamma.\Gamma' \vdash \mathcal{J}$ for the first statement, and on $\Theta; \Gamma \vdash \mathcal{J}$ for the second. ■

[7] Note that both implications are necessary in **(III)** because we can have rewrite rules both at the level of terms and sorts. On the other hand, **(II)** can be stated uniformly because, at the level of untyped terms, there is no distinction between terms and sorts.

Our theories also satisfy a substitution property, meaning that if $\Theta; \Gamma \vdash \mathcal{J}$ is derivable, then by applying any substitution typed by Γ or metavariable substitution typed by Θ the resulting judgment is still derivable. In order to state this property precisely, we first need to explain what it means to apply a substitution to a judgment $\mathcal{J}$. This is specified by the following table.

$\vdash \mathcal{J}$	$\vdash \mathcal{J}[\vec{v}]$	$\vdash \mathcal{J}[\mathbf{v}]_\Gamma$
$\vdash$	$\vdash$	$\vdash$
$\vdash T$ sort	$\vdash T[\vec{v}]$ sort	$\vdash T[\mathbf{v}]$ sort
$\vdash t : T$	$\vdash t[\vec{v}] : T[\vec{v}]$	$\vdash t[\mathbf{v}] : T[\mathbf{v}]$
$\vdash \vec{t} : \Delta$	$\vdash \vec{t}[\vec{v}] : \Delta$	$\vdash \mathsf{id}, \vec{t}[\mathbf{v}] : \Gamma.\Delta[\mathbf{v}]$
$\vdash \mathbf{t} : \Xi$	$\vdash \mathbf{t}[\vec{v}] : \Xi$	$\vdash \mathbf{t}[\mathbf{v}] : \Xi$

Note that in the case $\mathcal{J} = \vec{t} : \Delta$, taking $\mathcal{J}[\mathbf{v}]_\Gamma := \vec{t}[\mathbf{v}] : \Delta[\mathbf{v}]$ would not in general yield a well-formed judgment, given that $\mathbf{v}$ might introduce dangling variables in $\Delta[\mathbf{v}]$. Therefore, we need to prefix $\Delta[\mathbf{v}]$ with the context Γ of the substitution, and fill its positions with the identity id, yielding $\mathcal{J}[\mathbf{v}]_\Gamma := \mathsf{id}, \vec{t}[\mathbf{v}] : \Gamma.\Delta[\mathbf{v}]$.

Proposition 7.6 (Substitution property). *The following rules are admissible.*

$$\frac{\Theta; \Gamma \vdash \vec{v} : \Delta \qquad \Theta; \Delta \vdash \mathcal{J}}{\Theta; \Gamma \vdash \mathcal{J}[\vec{v}]} \qquad\qquad \frac{\Xi; \Gamma \vdash \mathbf{v} : \Theta \qquad \Theta; \Delta \vdash \mathcal{J}}{\Theta; \Gamma.\Delta[\mathbf{v}] \vdash \mathcal{J}[\mathbf{v}]_\Gamma}$$

Proof. For the proof to go through, we strengthen the first statement in the following way:

$$\frac{\Theta; \Gamma \vdash \vec{v} : \Delta \qquad \Theta; \Delta.\Gamma' \vdash \mathcal{J}}{\Theta; \Gamma.\Gamma'[\vec{v}] \vdash \mathcal{J}[\vec{v}, \mathsf{id}]}$$

Then both statements can be shown by induction, on $\Theta; \Delta.\Gamma' \vdash \mathcal{J}$ for the first statement, and $\Theta; \Delta \vdash \mathcal{J}$ for the second. Most cases follow directly from the induction hypothesis, the basic properties of substitution (Propositions 7.1 to 7.3) and, for the rule Conv, from the stability of conversion under substitution (Corollary 7.1). We show the key cases, and illustrate the other ones by some representative cases.

- Case Var of the first statement.

  $$x : T \in \Delta.\Gamma' \; \frac{\Theta; \Delta.\Gamma' \vdash}{\Theta; \Delta.\Gamma' \vdash x : T}$$

  We have either $x : T \in \Gamma'$ or $x : T \in \Delta$. In the first case, we apply the i.h. to get $\Theta; \Gamma.\Gamma'[\vec{v}] \vdash$ and then conclude with the variable rule. Otherwise, if $x : T \in \Delta$ then from $\Theta; \Gamma \vdash \vec{v} : \Delta$ we first get $\Theta; \Gamma \vdash v_x : T[\vec{v}']$, with $\vec{v}'$ being the prefix of $\vec{v}$ preceding v_x. Then, by i.h. we have $\Theta; \Gamma.\Gamma'[\vec{v}] \vdash$, so we can apply Proposition 7.5 to get $\Theta; \Gamma.\Gamma'[\vec{v}] \vdash v_x : T[\vec{v}']$. Because we have $T[\vec{v}'] = T[\vec{v}, \mathsf{id}]$ we are done.

- Case TmSym of the first statement.

$$\begin{array}{c} f(\Xi_1\ ;\ \Xi_2\ ;\ \vec{t} \equiv \vec{u}) : T \in \mathbb{T} \\ \vec{t}[\mathbf{t}_1,\mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1,\mathbf{t}_2] \end{array} \dfrac{\Theta;\Delta.\Gamma' \vdash \mathbf{t}_1,\mathbf{t}_2 : \Xi_1.\Xi_2 \qquad \Theta;\Delta.\Gamma' \vdash T[\mathbf{t}_1,\mathbf{t}_2]\ \mathsf{sort}}{\Theta;\Delta.\Gamma' \vdash f(\mathbf{t}_2) : T[\mathbf{t}_1,\mathbf{t}_2]}$$

  By i.h. we have $\Theta;\Gamma.\Gamma'[\vec{v}] \vdash (\mathbf{t}_1,\mathbf{t}_2)[\vec{v},\mathrm{id}] : \Xi_1.\Xi_2$ and $\Theta;\Gamma.\Gamma'[\vec{v}] \vdash T[\mathbf{t}_1,\mathbf{t}_2][\vec{v},\mathrm{id}]\ \mathsf{sort}$. From $\vec{t}[\mathbf{t}_1,\mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1,\mathbf{t}_2]$ we also get $\vec{t}[(\mathbf{t}_1,\mathbf{t}_2)[\vec{v},\mathrm{id}]] \equiv \vec{u}[(\mathbf{t}_1,\mathbf{t}_2)[\vec{v},\mathrm{id}]]$, and we also have $T[\mathbf{t}_1,\mathbf{t}_2][\vec{v},\mathrm{id}] = T[(\mathbf{t}_1,\mathbf{t}_2)[\vec{v},\mathrm{id}]]$, therefore we can derive $\Theta;\Gamma.\Gamma'[\vec{v}] \vdash f(\mathbf{t}_2[\vec{v},\mathrm{id}]) : T[(\mathbf{t}_1,\mathbf{t}_2)[\vec{v},\mathrm{id}]]$. Applying $T[(\mathbf{t}_1,\mathbf{t}_2)[\vec{v},\mathrm{id}]] = T[\mathbf{t}_1,\mathbf{t}_2][\vec{v},\mathrm{id}]$ once again, we conclude.

- Case ExtMSub of the first statement.

$$\dfrac{\Theta;\Delta.\Gamma' \vdash \mathbf{t} : \Xi \qquad \Theta;\Delta.\Gamma'.\Delta_{\mathsf{x}}[\mathbf{t}] \vdash t : T[\mathbf{t}]}{\Theta;\Delta.\Gamma' \vdash \mathbf{t}, \vec{x}_{\Delta}.t : (\Xi, \mathsf{x}\{\Delta_{\mathsf{x}}\} : T)}$$

  By i.h., $\Theta;\Gamma.\Gamma'[\vec{v}] \vdash \mathbf{t}[\vec{v},\mathrm{id}] : \Xi$ and $\Theta;\Gamma.(\Gamma'.\Delta_{\mathsf{x}}[\mathbf{t}])[\vec{v}] \vdash t[\vec{v},\mathrm{id}] : T[\mathbf{t}][\vec{v},\mathrm{id}]$. We have $(\Gamma'.\Delta_{\mathsf{x}}[\mathbf{t}])[\vec{v}] = \Gamma'[\vec{v}].\Delta_{\mathsf{x}}[\mathbf{t}[\vec{v},\mathrm{id}]]$ and $T[\mathbf{t}][\vec{v},\mathrm{id}] = T[\mathbf{t}[\vec{v},\mathrm{id}]]$, and so we also have $\Theta;\Gamma.\Gamma'[\vec{v}].\Delta_{\mathsf{x}}[\mathbf{t}[\vec{v},\mathrm{id}]] \vdash t[\vec{v},\mathrm{id}] : T[\mathbf{t}[\vec{v},\mathrm{id}]]$. Thus, we can build a derivation of $\Theta;\Gamma.\Gamma'[\vec{v}] \vdash \mathbf{t}[\vec{v},\mathrm{id}], \vec{x}_{\Delta_{\mathsf{x}}}.t[\vec{v},\mathrm{id}] : (\Xi, \mathsf{x}\{\Delta_{\mathsf{x}}\} : T)$, and because $(\mathbf{t}, \vec{x}_{\Delta_{\mathsf{x}}}.t)[\vec{v},\mathrm{id}] = \mathbf{t}[\vec{v},\mathrm{id}], \vec{x}_{\Delta_{\mathsf{x}}}.t[\vec{v},\mathrm{id}]$ we are done.

- Case MVar of the second statement.

$$\mathsf{x}\{\Delta'\} : T \in \Theta \ \dfrac{\Theta;\Delta \vdash \vec{t} : \Delta'}{\Theta;\Delta \vdash \mathsf{x}\{\vec{t}\} : T[\vec{t}]}$$

  By i.h. we have $\Xi;\Gamma.\Delta[\mathbf{v}] \vdash \mathrm{id}, \vec{t}[\mathbf{v}] : \Gamma.\Delta'[\mathbf{v}]$. Moreover, from $\Xi;\Gamma \vdash \mathbf{v} : \Theta$ we can deduce $\Xi;\Gamma.\Delta'[\mathbf{v}] \vdash \mathbf{v}_{\mathsf{x}} : T[\mathbf{v}]$, so by the substitution property for variable substitutions we get $\Xi;\Gamma.\Delta[\mathbf{v}] \vdash \mathbf{v}_{\mathsf{x}}[\mathrm{id}, \vec{t}[\mathbf{v}]] : T[\mathbf{v}][\mathrm{id}, \vec{t}[\mathbf{v}]]$, and because $\mathsf{x}\{\vec{t}\}[\mathbf{v}] = \mathbf{v}_{\mathsf{x}}[\mathrm{id}, \vec{t}[\mathbf{v}]]$ and $T[\mathbf{v}][\mathrm{id}, \vec{t}[\mathbf{v}]] = T[\vec{t}][\mathbf{v}]$ we are done

- Case EmptySub of the second statement.

$$\dfrac{\Theta;\Delta \vdash}{\Theta;\Delta \vdash \varepsilon : (\cdot)}$$

  By i.h. we have $\Xi;\Gamma.\Delta[\mathbf{v}] \vdash$. We can show $\Xi;\Gamma.\Delta[\mathbf{v}] \vdash \mathrm{id} : \Gamma$ and so we are done.

- Case ExtMSub of the second statement.

$$\dfrac{\Theta;\Delta \vdash \mathbf{t} : \Xi \qquad \Theta;\Delta.\Delta'[\mathbf{t}] \vdash t : T[\mathbf{t}]}{\Theta;\Delta \vdash \mathbf{t}, \vec{x}_{\Delta}.t : (\Xi, \mathsf{x}\{\Delta'\} : T)}$$

  By i.h. we have $\Xi;\Gamma.\Delta[\mathbf{v}] \vdash \mathbf{t}[\mathbf{v}] : \Xi$ and $\Xi;\Gamma.(\Delta.\Delta'[\mathbf{t}])[\mathbf{v}] \vdash t[\mathbf{v}] : T[\mathbf{t}][\mathbf{v}]$. We have $(\Delta.\Delta'[\mathbf{t}])[\mathbf{v}] = \Delta[\mathbf{v}].\Delta'[\mathbf{t}[\mathbf{v}]]$ and $T[\mathbf{t}][\mathbf{v}] = T[\mathbf{t}[\mathbf{v}]]$, therefore $\Xi;\Gamma.\Delta[\mathbf{v}].\Delta'[\mathbf{t}[\mathbf{v}]] \vdash t[\mathbf{v}] : T[\mathbf{t}[\mathbf{v}]]$. We can thus conclude $\Xi;\Gamma.\Delta[\mathbf{v}] \vdash \mathbf{t}[\mathbf{v}], \vec{x}.t[\mathbf{v}] : (\Xi, \mathsf{x}\{\Delta'\} : T)$. ■

The conversion rule in Figure 7.4 can be generalized to contexts in the following manner:

Proposition 7.7 (Conversion in context). *The following rules are admissible.*

$$\Delta \equiv \Delta' \frac{\Theta;\Gamma \vdash \vec{t} : \Delta \qquad \Theta;\Delta' \vdash}{\Theta;\Gamma \vdash \vec{t} : \Delta'} \qquad\qquad \Gamma \equiv \Delta \frac{\Theta;\Gamma \vdash \mathcal{J} \qquad \Theta;\Delta \vdash}{\Theta;\Delta \vdash \mathcal{J}}$$

Proof. We first show the first statement by induction on Δ, using Proposition 7.6. Then, for the proof of the second statement we instantiate the first with $\Theta;\Delta \vdash \mathrm{id} : \Delta$ to get $\Theta;\Delta \vdash \mathrm{id} : \Gamma$ and then conclude by applying Proposition 7.6 and using the fact that $\mathcal{J}[\mathrm{id}] = \mathcal{J}$. ■

Proposition 7.8 (Sorts are well-typed). *The following rule is admissible.*

$$\frac{\Theta;\Gamma \vdash t : T}{\Theta;\Gamma \vdash T \text{ sort}}$$

Proof. By case analysis on $\Theta;\Gamma \vdash t : T$, using Propositions 7.5 and 7.6 for case MVar and Proposition 7.5 for case Var. ■

Using Proposition 7.6, the premise for typing the sort in rule TmSym can now be dropped, as anticipated in Remark 7.6. In the following we allow ourselves to use this economic version of the rule TmSym without announcement.

Proposition 7.9 ("Economic" TmSym). *The following rule is admissible when $\mathbb{T}$ is well-typed.*

$$\begin{array}{c} f(\Xi_1 \,;\, \Xi_2 \,;\, \vec{t} \equiv \vec{u}) : T \in \mathbb{T} \\ \vec{t}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1, \mathbf{t}_2] \end{array} \frac{\Theta;\Gamma \vdash \mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2}{\Theta;\Gamma \vdash f(\mathbf{t}_2) : T[\mathbf{t}_1, \mathbf{t}_2]}$$

Proof. By well-typedness of the theory we have $\Xi_1.\Xi_2 \vdash T$ sort, therefore by Proposition 7.6 we get $\Theta;\Gamma \vdash T[\mathbf{t}_1, \mathbf{t}_2]$ sort, and thus $\Theta;\Gamma \vdash f(\mathbf{t}_2) : T[\mathbf{t}_1, \mathbf{t}_2]$ by rule TmSym. ■

Finally, we now conclude this chapter by showing that subject reduction can be reduced to a more local condition. Namely, we say that a rewrite rule $l \longmapsto r$ *preserves typing* if $\Theta;\Gamma \vdash l[\mathbf{t}] : T$ implies $\Theta;\Gamma \vdash r[\mathbf{t}] : T$ for all Θ, Γ, $\mathbf{t}$ and T, and if $\Theta;\Gamma \vdash l[\mathbf{t}]$ sort implies $\Theta;\Gamma \vdash r[\mathbf{t}]$ sort for all Θ, Γ and $\mathbf{t}$.

Proposition 7.10 (Subject reduction is equivalent to preservation of typing). *A well-typed theory satisfies subject reduction iff all its rewrite rules preserve typing.*

Proof. The direct implication is trivial, and for the reverse implication we show that preservation of typing implies the following statements:

- If $\Theta;\Gamma \vdash T$ sort and $T \longrightarrow T'$ then $\Theta;\Gamma \vdash T'$ sort

- If $\Theta; \Gamma \vdash t : T$ and $t \longrightarrow t'$ then $\Theta; \Gamma \vdash t' : T$
- If $\Theta; \Gamma \vdash \vec{t} : \Delta$ and $\Theta; \Delta \vdash$ and $\vec{t} \longrightarrow \vec{t}'$ then $\Theta; \Gamma \vdash \vec{t}' : \Delta$
- If $\Theta; \Gamma \vdash \mathbf{t} : \Xi$ and $\Xi \vdash$ and $\mathbf{t} \longrightarrow \mathbf{t}'$ then $\Theta; \Gamma \vdash \mathbf{t}' : \Xi$

The proof is by induction on the typing derivation, and by case analysis on the rewriting relation.

- Case $l[\mathbf{t}] \longrightarrow r[\mathbf{t}]$. The result follows directly from the assumption that all rewrite rules in $\mathbb{T}$ preserve typing.
- Case $f(\mathbf{t}) \longrightarrow f(\mathbf{t}')$ with $\mathbf{t} \longrightarrow \mathbf{t}'$, and where $f(\Xi)$ sort $\in \mathbb{T}$. By inversion of typing on $\Theta; \Gamma \vdash f(\mathbf{t})$ sort we have $\Theta; \Gamma \vdash \mathbf{t} : \Xi$, and because $\mathbb{T}$ is well-typed we have $\Xi \vdash$, so by i.h. we get $\Theta; \Gamma \vdash \mathbf{t}' : \Xi$, allowing us to conclude $\Theta; \Gamma \vdash f(\mathbf{t}')$ sort.
- Case $f(\mathbf{t}_2) \longrightarrow f(\mathbf{t}_2')$ with $\mathbf{t}_2 \longrightarrow \mathbf{t}_2'$, and where $f(\Xi_1 \mathbin{\mathbf{;}} \Xi_2 \mathbin{\mathbf{;}} \vec{t} \equiv \vec{u}) : U \in \mathbb{T}$. By inversion of typing on $\Theta; \Gamma \vdash f(\mathbf{t}_2) : T$ we have $\Theta; \Gamma \vdash \mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2$ and $\vec{t}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1, \mathbf{t}_2]$ and $T \equiv U[\mathbf{t}_1, \mathbf{t}_2]$. Because $\mathbb{T}$ is well-typed, we have $\Xi_1.\Xi_2 \vdash$, therefore by i.h. we get $\Theta; \Gamma \vdash \mathbf{t}_1, \mathbf{t}_2' : \Xi_1.\Xi_2$, and moreover we have $\vec{t}[\mathbf{t}_1, \mathbf{t}_2'] \equiv \vec{t}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1, \mathbf{t}_2']$. We can thus derive $\Theta; \Gamma \vdash f(\mathbf{t}_2') : U[\mathbf{t}_1, \mathbf{t}_2']$, and by Proposition 7.8 applied to $\Theta; \Gamma \vdash f(\mathbf{t}_2) : T$ we have $\Theta; \Gamma \vdash T$ sort, so we conclude by applying conversion with $U[\mathbf{t}_1, \mathbf{t}_2'] \equiv T$.
- Case $\mathsf{x}\{\vec{t}\} \longrightarrow \mathsf{x}\{\vec{t}'\}$ with $\vec{t} \longrightarrow \vec{t}'$, and where $\mathsf{x}\{\Delta\} : U \in \Theta$. By inversion of typing on $\Theta; \Gamma \vdash \mathsf{x}\{\vec{t}\} : T$ we obtain $\Theta; \Gamma \vdash \vec{t} : \Delta$ and $T \equiv U[\vec{t}]$. From $\Theta \vdash$ we can extract a derivation of $\Theta'; \Delta \vdash$ for some $\Theta' \sqsubseteq \Theta$, which by Proposition 7.5 gives $\Theta; \Delta \vdash$. Therefore, we can apply the i.h. to get $\Theta; \Gamma \vdash \vec{t}' : \Delta$, and so $\Theta; \Gamma \vdash \mathsf{x}\{\vec{t}'\} : U[\vec{t}']$. We thus have $U[\vec{t}'] \equiv T$, and by Proposition 7.8 applied to $\Theta; \Gamma \vdash \mathsf{x}\{\vec{t}\} : T$ we have $\Theta; \Gamma \vdash T$ sort, allowing us to conclude $\Theta; \Gamma \vdash \mathsf{x}\{\vec{t}'\} : T$ using the conversion rule.
- Case $\vec{u}, t \longrightarrow \vec{t}'$. By inversion on $\Theta; \Gamma \vdash \vec{u}, t : \Delta$ we get $\Delta = \Delta', x : U$ and $\Theta; \Gamma \vdash \vec{u} : \Delta'$ and $\Theta; \Gamma \vdash t : U[\vec{u}]$. Moreover, from $\Theta; \Delta \vdash$ we get $\Theta; \Delta' \vdash U$ sort.
  - Case $\vec{t}' = \vec{u}', t$ with $\vec{u} \longrightarrow \vec{u}'$. Then by i.h. we get $\Theta; \Gamma \vdash \vec{u}' : \Delta'$. By Proposition 7.6 with $\Theta; \Delta' \vdash U$ sort we get $\Theta; \Gamma \vdash U[\vec{u}']$ sort, so we can apply conversion to $\Theta; \Gamma \vdash t : U[\vec{u}]$ to get $\Theta; \Gamma \vdash t : U[\vec{u}']$, allowing us to conclude $\Theta; \Gamma \vdash \vec{u}', t : (\Delta', x : U)$.
  - Case $\vec{t}' = \vec{u}, t'$ with $t \longrightarrow t'$. Then by i.h. we get $\Theta; \Gamma \vdash t' : U[\vec{u}]$, allowing us to conclude $\Theta; \Gamma \vdash \vec{u}, t' : (\Delta', x : U)$.
- Case $\mathbf{u}, \vec{x}.t \longrightarrow \mathbf{t}'$. By inversion on $\Theta; \Gamma \vdash \mathbf{u}, \vec{x}.t : \Xi$ we get $\Xi = \Xi', \mathsf{x}\{\Delta\} : U$ and $\Theta; \Gamma \vdash \mathbf{u} : \Xi'$ and $\Theta; \Gamma.\Delta[\mathbf{u}] \vdash t : U[\mathbf{u}]$. Moreover, from $\Xi \vdash$ we get $\Xi'; \Delta \vdash U$ sort.

- Subcase $\mathbf{t}' = \mathbf{u}', \vec{x}.t$ with $\mathbf{u} \longrightarrow \mathbf{u}'$. Then by i.h. we get $\Theta; \Gamma \vdash \mathbf{u}' : \Xi'$. By Proposition 7.6 with $\Xi'; \Delta \vdash U$ sort we get $\Theta; \Gamma.\Delta[\mathbf{u}'] \vdash U[\mathbf{u}']$ sort, so we can apply conversion and Proposition 7.7 to $\Theta; \Gamma.\Delta[\mathbf{u}] \vdash t : U[\mathbf{u}]$ to get $\Theta; \Gamma.\Delta[\mathbf{u}'] \vdash t : U[\mathbf{u}']$, allowing us to conclude $\Theta; \Gamma \vdash \mathbf{u}', \vec{x}.t : (\Xi', \mathsf{x}\{\Delta\} : U)$.
- Subcase $\mathbf{t}' = \mathbf{u}, \vec{x}.t'$ with $t \longrightarrow t'$. By i.h. we get $\Theta; \Gamma.\Delta[\mathbf{u}] \vdash t' : U[\mathbf{u}]$, and so we conclude $\Theta; \Gamma \vdash \mathbf{u}, \vec{x}.t' : (\Xi', \mathsf{x}\{\Delta\} : U)$. ■

Example 7.8. To show that $\mathbb{T}_{\lambda\Pi}$ satisfies subject reduction, we can apply Proposition 7.10 and reduce this to verifying that the rule $@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$ preserves typing. By inversion of typing, it is clear that $\Theta; \Gamma \vdash @(\lambda(x.\mathsf{t}\{x\}), \mathsf{u})[x.t, u]$ sort can never hold, so we only need to verify that $\Theta; \Gamma \vdash @(\lambda(x.\mathsf{t}\{x\}), \mathsf{u})[x.t, u] : T$ implies $\Theta; \Gamma \vdash \mathsf{t}\{\mathsf{u}\}[x.t, u] : T$, for all Θ, Γ, T, t and u.

We have $@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u})[x.t, u] = @(\lambda(x.t), u)$ and $\mathsf{t}\{\mathsf{u}\}[x.t, u] = t[\mathrm{id}, u]$, so we start by applying inversion of typing to $\Theta; \Gamma \vdash @(\lambda(x.t), u) : T$ to get

$$\Theta; \Gamma \vdash A : \mathrm{Ty} \qquad\qquad \Theta; \Gamma, x : \mathrm{Tm}(A) \vdash B : \mathrm{Ty}$$

$$\Theta; \Gamma \vdash \lambda(x.t) : \mathrm{Tm}(\Pi(A, x.B)) \qquad\qquad \Theta; \Gamma \vdash u : \mathrm{Tm}(A)$$

for some A, B with $T \equiv \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})[A, x.B, t, u] = \mathrm{Tm}(B[\mathrm{id}, u])$. By inversion again, but this time on $\Theta; \Gamma \vdash \lambda(x.t) : \mathrm{Tm}(\Pi(A, x.B))$, we then have

$$\Theta; \Gamma \vdash A' : \mathrm{Ty} \qquad \Theta; \Gamma, x : \mathrm{Tm}(A') \vdash B' : \mathrm{Ty} \qquad \Theta; \Gamma, x : \mathrm{Tm}(A') \vdash t : \mathrm{Tm}(B')$$

for some A', B' with $\mathrm{Tm}(\Pi(A, x.B)) \equiv \mathrm{Tm}(\Pi(A', x.B'))$. Using confluence and the fact that no rewrite rule is headed by Tm or Π, it is easy to see that the pattern $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$ is *injective*, meaning that $\mathrm{Tm}(\Pi(A, x.B)) \equiv \mathrm{Tm}(\Pi(A', x.B'))$ implies $A \equiv A'$ and $B \equiv B'$.

Now we can conclude by applying some of the meta-theorems we saw in this subsection, that crucially *do not* rely on subject reduction — which would otherwise incur a circularity in our reasoning. More precisely, from the above derivations we can show $\Theta; \Gamma, x : \mathrm{Tm}(A) \vdash$ and $\Theta; \Gamma, x : \mathrm{Tm}(A) \vdash \mathrm{Tm}(B)$ sort, so starting from $\Theta; \Gamma, x : \mathrm{Tm}(A') \vdash t : \mathrm{Tm}(B')$ we can apply Proposition 7.7 and conversion to obtain $\Theta; \Gamma, x : \mathrm{Tm}(A) \vdash t : \mathrm{Tm}(B)$. We can also show $\Theta; \Gamma \vdash \mathrm{id}, u : (\Gamma, x : \mathrm{Tm}(A))$, so by Proposition 7.6 we then get $\Theta; \Gamma \vdash t[\mathrm{id}, u] : \mathrm{Tm}(B[\mathrm{id}, u])$. Finally, by applying Proposition 7.8 to $\Theta; \Gamma \vdash @(\lambda(x.t), u) : T$ we get $\Theta; \Gamma \vdash T$ sort, so by applying conversion with $\mathrm{Tm}(B[\mathrm{id}, u]) \equiv T$ we conclude $\Theta; \Gamma \vdash t[\mathrm{id}, u] : T$.

Importantly, note that the above verification only relied on two properties: (1) that the theory is well-typed, and (2) that the pattern $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$ is injective. This means that the rule $@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}$ actually preserves typing in *any* extension of $\mathbb{T}_{\lambda\Pi}$ satisfying these two properties, an observation that will be relevant when considering extensions of this theory in Chapter 9. □

Remark 7.8. It is easy to see that the procedure of Example 7.8 could be automated, as done already in the case of Dedukti [Sai15b, Bla20]. We leave this for future work. □

Chapter 8

Generic Bidirectional Typing

In Chapter 7 we specified the theories of our framework, as well as the type system they define. However, as discussed in Chapter 6, the omission of some arguments in the syntax means that this system is not algorithmic, and when building derivations one is obliged to guess the missing information. The goal of this chapter is to refine the notion of theory seen in the last chapter into *bidirectional theories*, for which we develop bidirectional typing *generically*. More precisely, for each bidirectional theory, we define a bidirectional type system and show it to be not only equivalent to the type system of Figure 7.4 but also to be decidable for strongly normalizing theories.

We start the chapter by defining our notion of bidirectional theory, and then discussing the problem of matching modulo, which is needed for recovering missing arguments. We then continue by introducing the bidirectional syntax, over which the bidirectional type system is defined, and defining the bidirectional system itself. We conclude the chapter with the definition of the *valid* bidirectional theories, and the proofs of correctness and decidability of the bidirectional type system.

8.1 Bidirectional theories

When specifying the theories in Section 7.2 we allowed term rules to omit some of their arguments without worrying how they could be algorithmically recovered. We fix this by defining *bidirectional theories* $\mathbb{T}^\flat$, in which term rules are required to be of a certain shape.

Recall from Chapter 6 that the way in which missing arguments can be recovered in bidirectional typing is deeply linked with whether a symbol is a *constructor* or a *destructor*. In order to capture this distinction, let us suppose that symbols $f, g, \dots$ are partitioned as constructors c and destructors d. We then update our notational convention and write destructor names in orange, so that they stand apart from constructor names, which are still written in blue. Then, the bidirectional theories are defined by Figure 8.1, where, given a bidirectional theory $\mathbb{T}^\flat$, we write $\mathbb{T}$ for its underlying regular theory, obtained by mapping $d(\Xi_1 \mathbin{;} \mathsf{x} :_{\mathrm{p}} T \mathbin{;} \Xi_2) : U$ into $d(\Xi_1 \mathbin{;} \mathsf{x} : T, \Xi_2 \mathbin{;} \varepsilon \equiv \varepsilon) : U$, and keeping all the other rules as they are.

$$
\begin{aligned}
\boxed{\mathsf{Thy}^\flat} \ni \mathbb{T}^\flat ::= {} & \mid \cdot \\
& \mid \mathbb{T}^\flat,\ c(\Xi \in \mathsf{MCtx}_{|\mathbb{T}|})\ \mathsf{sort} \\
& \mid \mathbb{T}^\flat,\ c(\Xi_1 \in \mathsf{MCtx}_{|\mathbb{T}|}\ ;\ \Xi_2 \in \mathsf{MCtx}_{|\mathbb{T}|}\ |\Xi_1|\ ; \\
& \qquad \vec{t} \in \mathsf{Tm}_{|\mathbb{T}|}\ |\Xi_1.\Xi_2|\ \gamma \equiv \vec{u} \in \mathsf{Tm}_{|\mathbb{T}|}\ |\Xi_1.\Xi_2|\ \gamma) : T \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ |\Xi_1|\ (\cdot) \\
& \mid \mathbb{T}^\flat,\ d(\Xi_1 \in \mathsf{MCtx}_{|\mathbb{T}|}\ ;\ \mathsf{x} :_{\mathrm{p}} T \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ |\Xi_1|\ (\cdot)\ ; \\
& \qquad \Xi_2 \in \mathsf{MCtx}_{|\mathbb{T}|}\ (|\Xi_1|, \mathsf{x})) : U \in \mathsf{Tm}_{|\mathbb{T}|}\ (|\Xi_1|, \mathsf{x}, |\Xi_2|)\ (\cdot) \\
& \mid \mathbb{T}^\flat,\ \theta \Vdash l \in \mathsf{Tm}^{\mathsf{P}}_{|\mathbb{T}|}\ \theta\ (\cdot) \longmapsto r \in \mathsf{Tm}_{|\mathbb{T}|}\ \theta\ (\cdot) \qquad \text{with } l \text{ not a metavariable}
\end{aligned}
$$

Figure 8.1: Definition of bidirectional theories

Remark 8.1. Note that in sort rules we impose the associated symbol to be a constructor. It would also be reasonable to introduce a specific class of symbols for sorts, but, as it turns out, considering them to be constructors seems to make the developments of this chapter simpler. □

Constructor rules

In Figure 8.1, schematic rules of the form

$$c(\Xi_1 \in \mathsf{MCtx}\ ;\ \Xi_2 \in \mathsf{MCtx}\ |\Xi_1|\ ;\ \vec{t} \in \mathsf{Tm}\ |\Xi_1.\Xi_2|\ \gamma \equiv \vec{u} \in \mathsf{Tm}\ |\Xi_1.\Xi_2|\ \gamma) : T \in \mathsf{Tm}^{\mathsf{P}}\ |\Xi_1|\ (\cdot)$$

are called *constructor rules*. Recall from Chapter 6 that in bidirectional typing constructors support type-checking, so that the missing information can be recovered from the sort given as input. In order to ensure that this is possible, the sort T of a constructor rule is required to be a pattern over the metavariables of the erased arguments Ξ_1 — we will then see in detail how this allows arguments to be recovered in Section 8.2.

Two examples of constructor rules are the ones for Π and λ in $\mathbb{T}_{\lambda\Pi}$:

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty}}{\Pi(\mathsf{A}, x.\mathsf{B}\{x\}) : \mathrm{Ty}} \qquad \frac{\begin{array}{c}\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{B}\{x\})\end{array}}{\lambda(x.\mathsf{t}\{x\}) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))}$$

The requirement that the sort of the rule is a pattern over $|\Xi_1|$ can however be problematic in some cases, for instance when trying to define the constructor refl for the equality type, and the constructor cons for the type of vectors.

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad \mathsf{a} : \mathrm{Tm}(\mathsf{A})}{\mathrm{refl} : \mathrm{Tm}(\mathrm{Eq}(\mathsf{A}, \mathsf{a}, \mathsf{a}))} \qquad \frac{\mathsf{A} : \mathrm{Ty} \qquad \mathsf{n} : \mathrm{Tm}(\mathrm{Nat}) \qquad \mathsf{t} : \mathrm{Tm}(\mathsf{A}) \qquad \mathsf{l} : \mathrm{Tm}(\mathrm{Vec}(\mathsf{A}, \mathsf{n}))}{\mathrm{cons}(\mathsf{n}, \mathsf{t}, \mathsf{l}) : \mathrm{Tm}(\mathrm{Vec}(\mathsf{A}, \mathrm{S}(\mathsf{n})))}$$

In the case of refl, the metavariable a occurs *non-linearly* in its sort $\mathrm{Tm}(\mathrm{Eq}(\mathsf{A},\mathsf{a},\mathsf{a}))$, which is therefore not a pattern in our sense. In the case of cons its sort actually *is* a pattern, however because it contains the metavariable n we would need to omit it in the syntax. This in principle can appear to be a good thing, after all the goal of bidirectional typing is precisely to remove annotations that are not needed. However when writing the reduction rules associated with the eliminator for vectors we realize that n actually *is* needed. Indeed, the argument n is *computationally relevant*, meaning that the result of a computation might depend on it, and therefore it cannot be erased — see the discussion in Section 9.2.

This is where equational hypotheses come in handy, as they allow us to rephrase the previous rules in the following manner, fitting the format required for constructor rules. The technique of moving from the previous presentation of the rules to the one using equational constraints is known as *fording* [McB00, CDMM10] and will be discussed in more details in Section 9.2.

$$\frac{\begin{array}{cc}\mathsf{A} : \mathrm{Ty} & \mathsf{a} : \mathrm{Tm}(\mathsf{A}) \\ \mathsf{b} : \mathrm{Tm}(\mathsf{B}) & \mathsf{a} \equiv \mathsf{b}\end{array}}{\mathrm{refl} : \mathrm{Tm}(\mathrm{Eq}(\mathsf{A},\mathsf{a},\mathsf{b}))} \qquad \frac{\begin{array}{ccc}\mathsf{A} : \mathrm{Ty} & \mathsf{m} : \mathrm{Tm}(\mathrm{Nat}) & \mathsf{n} : \mathrm{Tm}(\mathrm{Nat}) \\ \mathsf{t} : \mathrm{Tm}(\mathsf{A}) & \mathsf{l} : \mathrm{Tm}(\mathrm{Vec}(\mathsf{A},\mathsf{n})) & \mathsf{m} \equiv \mathrm{S}(\mathsf{n})\end{array}}{\mathrm{cons}(\mathsf{n},\mathsf{t},\mathsf{l}) : \mathrm{Tm}(\mathrm{Vec}(\mathsf{A},\mathsf{m}))}$$

Destructor rules

In opposition to constructors, in bidirectional typing destructors support *type inference*, and the missing arguments are instead recovered by inferring the first non-erased argument, which is called the *principal argument*. This leads to *destructor rules* of the form

$$d(\Xi_1 \in \mathsf{MCtx}\ ;\ \mathsf{x} :_{\mathrm{p}} T \in \mathsf{Tm}^{\mathsf{P}}\ |\Xi_1|\ (\cdot)\ ;\ \Xi_2 \in \mathsf{MCtx}\ (|\Xi_1|, \mathsf{x})) : U \in \mathsf{Tm}\ (|\Xi_1|, \mathsf{x}, |\Xi_2|)\ (\cdot)$$

Note that because the missing arguments in Ξ_1 are now recovered from the sort of the principal argument $\mathsf{x} :_{\mathrm{p}} T$, then its sort most be a pattern over the metavariables of Ξ_1.

The prime example of a destructor rule is the following one for application. Note that writing destructor names in orange and annotating the p in $\mathsf{x} :_{\mathrm{p}} T$ is helpful here so that we can easily "parse" the informal rules into the formal notation.

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} \qquad \mathsf{t} :_{\mathrm{p}} \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \qquad \mathsf{u} : \mathrm{Tm}(\mathsf{A})}{@(\mathsf{t},\mathsf{u}) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\})}$$

Remark 8.2. One may wonder why we impose the associated context of the principal argument to be always empty. If we had principal arguments of the form $\mathsf{x}\{\Gamma\} :_{\mathrm{p}} T$ instead of $\mathsf{x} :_{\mathrm{p}} T$, we would have to extend the current context by Γ before inferring x, yet Γ may make reference to the erased arguments, which are only recovered *after* the sort of the principal argument is inferred. □

Remark 8.3. As a simplification, here we consider destructor rules without equational premises, as we do not know of any interesting examples in which they would be needed. Nevertheless, modifying the developments of this chapter to allow for destructor rules with equations would be straightforward. □

Example 8.1. As we have just seen, all of the term rules of $\mathbb{T}_{\lambda\Pi}$ can be seen as either constructor or destructor rules, and so we obtain the following bidirectional theory:

$$\begin{aligned}
&\mathrm{Ty}(\cdot)\text{ sort},\quad \mathrm{Tm}(\mathsf{A} : \mathrm{Ty})\text{ sort},\quad \Pi(\cdot\ ;\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\varepsilon \equiv \varepsilon) : \mathrm{Ty}, && (\mathbb{T}^{\flat}_{\lambda\Pi})\\
&\lambda(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\mathsf{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B}\{x\})\ ;\varepsilon \equiv \varepsilon) : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})),\\
&@(\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty}\ ;\mathsf{t} :_{\mathrm{p}} \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))\ ;\mathsf{u} : \mathrm{Tm}(\mathsf{A})) : \mathrm{Tm}(\mathsf{B}\{\mathsf{u}\}),\\
&@(\lambda(x.\mathsf{t}\{x\}), \mathsf{u}) \longmapsto \mathsf{t}\{\mathsf{u}\}
\end{aligned}$$
□

8.2 Matching modulo

Suppose we want to type $@(t, u)$ by first inferring the sort of t, yielding T. We know that the sort of the principal argument in the rule for $@$ is the pattern $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$, so we could hope to recover A and B by matching T against this pattern. However, because of the conversion rule, in dependent type theories we cannot expect T to be syntactically equal to an instance of this pattern, but only convertible to it. Therefore, our goal is instead to find A and B satisfying $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))[A, x.B] \equiv T$. This shows that the process of recovering missing arguments in bidirectional typing is actually an instance of *matching modulo* — a connection that apparently had not been made explicit before in the bidirectional typing literature.

In order to solve matching modulo problems, we define in Figure 8.2 an inference system which, given a pattern t and a term u, tries to compute a metavariable substitution $\mathbf{v}$ such that $t[\mathbf{v}] \equiv u$. Here we write $t \longrightarrow^{\mathsf{h}} u$ when $t \longrightarrow^{*} u$ and u is *head-normal*, meaning that for all u' with $u \longrightarrow^{*} u'$, no rewrite rule can be applied to the head of u'.

Correctness of matching modulo

Let us now establish the correctness of this inference system in three steps, starting by its soundness.

Proposition 8.1 (Soundness of matching)**.**

- *If* $t \prec u \rightsquigarrow \mathbf{v}$ *then* $u \longrightarrow^{*} t[\mathbf{v}]$.
- *If* $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}$ *then* $\mathbf{u} \longrightarrow^{*} \mathbf{t}[\mathbf{v}]$.

Proof. By induction on the derivation. ■

$$\boxed{t \prec u \rightsquigarrow \mathbf{v} \quad (t \in \mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma;\ u \in \mathsf{Tm}\ (\cdot)\ \delta.\gamma;\ \mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta)}$$

$$u \longrightarrow^{\mathsf{h}} f(\mathbf{u})\,\frac{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}}{f(\mathbf{t}) \prec u \rightsquigarrow \mathbf{v}} \qquad\qquad \frac{}{\mathsf{x}\{\mathsf{id}_{\gamma}\} \prec u \rightsquigarrow \vec{x}_{\gamma}.u}$$

$$\boxed{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v} \quad (\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi;\ \mathbf{u} \in \mathsf{MSub}\ (\cdot)\ \delta.\gamma\ \xi;\ \mathbf{v} \in \mathsf{Sub}\ (\cdot)\ \delta\ \theta)}$$

$$\frac{}{\varepsilon \prec \varepsilon \rightsquigarrow \varepsilon} \qquad\qquad \frac{\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}_1 \qquad t \prec u \rightsquigarrow \mathbf{v}_2}{\mathbf{t}, \vec{x}.t \prec \mathbf{u}, \vec{x}.u \rightsquigarrow \mathbf{v}_1, \mathbf{v}_2}$$

Figure 8.2: Inference system for matching modulo

Is our inference system for matching modulo always complete? Is easy to see that trying to match the term $@(\lambda(x.x), y)$ against the pattern $@(\mathsf{t}, \mathsf{u})$ fails, even though the former is an instance of the latter. The problem here is that rewriting an instance of $@(\mathsf{t}, \mathsf{u})$ can lead to a term that is not an instance of this pattern anymore. While completeness might be recovered in such cases using the technique of *narrowing* [BS01, MH94, Pre94], the procedure becomes more complex, less efficient and the solutions one obtain are in general not unique modulo conversion. For instance, the metavariable substitutions $\lambda(x.x), y \in \mathsf{MSub}\ (\cdot)\ (y)\ (\mathsf{t}, \mathsf{u})$ and $\lambda(x.y), y \in \mathsf{MSub}\ (\cdot)\ (y)\ (\mathsf{t}, \mathsf{u})$ are both solutions to the above matching problem, yet they are not convertible. Therefore, we instead chose to impose an extra condition, ruling out patterns like $@(\mathsf{t}, \mathsf{u})$, and which, in the context of bidirectional typing, is always satisfied in practice.

Recall from rewriting theory that a (term or metavariable substitution) pattern e *overlaps* with a term pattern t_2 when, for some subterm t_1 of e not of the form $\mathsf{x}\{\vec{x}\}$, the terms t_1 and t_2 *unify*, meaning that we have $t_1[\mathbf{t}_1] = t_2[\mathbf{t}_2]$ for some metavariable substitutions $\mathbf{t}_1$ and $\mathbf{t}_2$. We then say that a (term or metavariable substitution) pattern e is *rigid* when it overlaps with the left-hand side of no rewrite rule. For instance, in $\mathbb{T}^{\flat}_{\lambda\Pi}$, the pattern $@(\mathsf{t}, \mathsf{u})$ is not rigid because its (non-proper) subterm $@(\mathsf{t}, \mathsf{u})$ unifies with the left-hand side of the β-rule. In contrast, no non-metavariable subterm of $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$ unifies with a rewrite rule left-hand side, and so this pattern is rigid.

We now show that the system of Figure 8.2 is indeed complete provided that the pattern we are matching against is rigid and the rewrite system is confluent. To do this, we first show the following technical result, ensuring that rewriting an instance of a rigid pattern leads to another instance of it.

Lemma 8.1 (Rewriting a rigid pattern).

- *Suppose that* $t \in \mathsf{Tm}^{\mathsf{P}}\ \xi\ \delta$ *is rigid. If for some* $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ *we have* $t[\mathbf{v}] \longrightarrow t'$ *then* $t' = t[\mathbf{v}']$ *for some* $\mathbf{v}' \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ *with* $\mathbf{v} \longrightarrow \mathbf{v}'$*.*

- *Suppose that* $\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \xi\ \delta\ \theta$ *is rigid. If for some* $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ *we have* $\mathbf{t}[\mathbf{v}] \longrightarrow \mathbf{t}'$ *then* $\mathbf{t}' = \mathbf{t}[\mathbf{v}']$ *for some* $\mathbf{v}' \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ *with* $\mathbf{v} \longrightarrow \mathbf{v}'$.

Proof. By mutual induction on t and $\mathbf{t}$.

- Case $t = \mathsf{x}\{\mathsf{id}_\delta\}$. We therefore have $\xi = \mathsf{x}\{\delta\}$. By taking $\mathbf{v}' := (\vec{x}_\delta.t') \in \mathsf{MSub}\ (\cdot)\ \gamma\ (\mathsf{x}\{\delta\})$ we get $\mathsf{x}\{\mathsf{id}_\delta\}[\mathbf{v}'] = t'$ and $\mathbf{v} \longrightarrow \mathbf{v}'$.
- Case $t = f(\mathbf{t})$. We have $f(\mathbf{t})[\mathbf{v}] \longrightarrow t'$, and because $f(\mathbf{t})$ is rigid, it follows that for no rule $l \longmapsto r$ and $\mathbf{v}'$ we can have $f(\mathbf{t})[\mathbf{v}] = l[\mathbf{v}']$. Therefore, the reduction step $f(\mathbf{t})[\mathbf{v}] \longrightarrow t'$ cannot happen in the head. We thus have $t' = f(\mathbf{t}')$ with $\mathbf{t}[\mathbf{v}] \longrightarrow \mathbf{t}'$. By the i.h., we get $\mathbf{t}' = \mathbf{t}[\mathbf{v}']$ for some $\mathbf{v}' \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ with $\mathbf{v} \longrightarrow \mathbf{v}'$, and thus $t' = f(\mathbf{t}') = f(\mathbf{t})[\mathbf{v}]$.
- Case $\mathbf{t} = \varepsilon$. Impossible.
- Case $\mathbf{t} = \mathbf{u}, \vec{x}.u$. Therefore, θ is of the form $\theta_0, \mathsf{x}\{\gamma_0\}$ and ξ is of the form $\xi_1.\xi_2$, and we have $\mathbf{u} \in \mathsf{MSub}^{\mathsf{P}}\ \xi_1\ \delta\ \theta_0$ and $t \in \mathsf{Tm}^{\mathsf{P}}\ \xi_2\ \delta.\gamma_0$. Moreover, we must have $\mathbf{t}'$ of the form $\mathbf{u}', \vec{x}.u'$. Let us split $\mathbf{v}$ into $\mathbf{v}_1.\mathbf{v}_2$ with $\mathbf{v}_i \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi_i$, so that $\mathbf{u}[\mathbf{v}_1], \vec{x}.u[\mathbf{v}_2] \longrightarrow \mathbf{u}', \vec{x}.u'$.
  - Subcase $\mathbf{u}[\mathbf{v}_1] \longrightarrow \mathbf{u}'$. We have $u' = u[\mathbf{v}_2]$, and by i.h. we get $\mathbf{u}' = \mathbf{u}[\mathbf{v}_1']$ with $\mathbf{v}_1' \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi_1$ and $\mathbf{v}_1 \longrightarrow \mathbf{v}_1'$. Thus $\mathbf{t}' = \mathbf{u}[\mathbf{v}_1'], \vec{x}.u[\mathbf{v}_2] = \mathbf{t}[\mathbf{v}_1', \mathbf{v}_2]$ with $\mathbf{v}_1, \mathbf{v}_2 \longrightarrow \mathbf{v}_1', \mathbf{v}_2$.
  - Subcase $u[\mathbf{v}_2] \longrightarrow u'$. We have $\mathbf{u}' = \mathbf{u}[\mathbf{v}_1]$, and by i.h. we get $u' = u[\mathbf{v}_2']$ with $\mathbf{v}_2' \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi_2$ and $\mathbf{v}_2 \longrightarrow \mathbf{v}_2'$. Thus $\mathbf{t}' = \mathbf{u}[\mathbf{v}_1], \vec{x}.u[\mathbf{v}_2'] = \mathbf{t}[\mathbf{v}_1, \mathbf{v}_2']$ with $\mathbf{v}_1, \mathbf{v}_2 \longrightarrow \mathbf{v}_1, \mathbf{v}_2'$. ■

Remark 8.4. Lemma 8.1 might feel familiar to readers knowledgeable in rewriting theory: it is very similar to the main auxiliary lemma used in proofs of confluence by orthogonality [MN98, BKdVT03], except that there one replaces ⟶ by *developments*. □

We can now show the completeness of matching:[1]

Proposition 8.2 (Completeness of matching)**.** *Suppose that the underlying theory is confluent and let* $\mathbf{v} \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta$.

- *If* $t \in \mathsf{Tm}^{\mathsf{P}}\ \theta\ \gamma$ *is rigid and* $t[\mathbf{v}] \equiv u$ *then* $t \prec u \rightsquigarrow \mathbf{v}'$ *for some* $\mathbf{v}' \equiv \mathbf{v}$.
- *If* $\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ \theta\ \gamma\ \xi$ *is rigid and* $\mathbf{t}[\mathbf{v}] \equiv \mathbf{u}$ *then* $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}'$ *for some* $\mathbf{v}' \equiv \mathbf{v}$.

[1] Combined with Lemma 8.6, this implies in particular the unicity of solutions modulo conversion, provided that the pattern is rigid and the rewrite system is confluent.

Proof. By induction on the pattern, the only interesting case being when $t = f(\mathbf{t})$. In this case, by confluence we have $u \longrightarrow^* u' \ {}^*\!\!\longleftarrow f(\mathbf{t})[\mathbf{v}]$, so by iterating Lemma 8.1 we obtain $u' = f(\mathbf{t})[\mathbf{v}']$ for some $\mathbf{v}' \in \mathsf{MSub}\ (\cdot)\ \delta\ \theta$ with $\mathbf{v} \longrightarrow^* \mathbf{v}'$, implying in particular $\mathbf{t}[\mathbf{v}'] \equiv \mathbf{t}[\mathbf{v}]$. Moreover, it is easy to see that $f(\mathbf{t})[\mathbf{v}']$ is head-normal: given any u'' such that $f(\mathbf{t})[\mathbf{v}'] \longrightarrow^* u''$, by iterating Lemma 8.1 again we deduce that u'' is of the form $f(\mathbf{t})[\mathbf{v}'']$, so because $f(\mathbf{t})$ is rigid it follows that u'' does not match any rewrite rule at the head. To conclude it suffices to note that, because $f(\mathbf{t})$ is rigid, then $\mathbf{t}$ also is, so by the i.h. we get $\mathbf{t} < \mathbf{t}[\mathbf{v}'] \rightsquigarrow \mathbf{v}''$ for some $\mathbf{v}'' \equiv \mathbf{v}$. ■

A direct consequence of soundness and completeness of matching is the following corollary, which will be useful in the proof of Proposition 8.3.

Corollary 8.1 (Matching respects conversion). *Suppose that underlying theory is confluent.*

- *If t is rigid and $t < u \rightsquigarrow \mathbf{v}$ and $u \equiv u'$ then $t < u' \rightsquigarrow \mathbf{v}'$ for some $\mathbf{v}' \equiv \mathbf{v}$.*
- *If $\mathbf{t}$ is rigid and $\mathbf{t} < \mathbf{u} \rightsquigarrow \mathbf{v}$ and $\mathbf{u} \equiv \mathbf{u}'$ then $\mathbf{t} < \mathbf{u}' \rightsquigarrow \mathbf{v}'$ for some $\mathbf{v}' \equiv \mathbf{v}$.*

Finally, we establish that matching modulo is decidable when the expression e from which we extract the substitution is *strongly normalizing* (often abbreviated as s.n.), meaning that all reduction sequences issuing from e are finite.

Proposition 8.3 (Decidability of matching). *Suppose that the underlying theory is confluent.*

- *If u is strongly normalizing then $\exists \mathbf{v}.\ t < u \rightsquigarrow \mathbf{v}$ is decidable for all t rigid.*
- *If $\mathbf{u}$ is strongly normalizing then $\exists \mathbf{v}.\ \mathbf{t} < \mathbf{u} \rightsquigarrow \mathbf{v}$ is decidable for all $\mathbf{t}$ rigid.*

Proof. By induction on the pattern. We show the only interesting case, when $t = f(\mathbf{t})$. Because u is strongly normalizing, we can use any reduction strategy to compute a head-normal form u' for u.

If u' is not headed by f then $\exists \mathbf{v}.\ t < u \rightsquigarrow \mathbf{v}$ cannot hold. Indeed, this would imply $u \longrightarrow^* f(\mathbf{u})$ for some $\mathbf{u}$, so by confluence and the fact that $f(\mathbf{u})$ and u' are head-normal we would be able to show that u' is headed by f, contradiction.

If u' is of the form $f(\mathbf{u})$, then by i.h. we can decide $\exists \mathbf{v}.\ \mathbf{t} < \mathbf{u} \rightsquigarrow \mathbf{v}$. If this holds, then it follows that $\exists \mathbf{v}.\ t < u \rightsquigarrow \mathbf{v}$ holds. If $\exists \mathbf{v}.\ \mathbf{t} < \mathbf{u} \rightsquigarrow \mathbf{v}$ does not hold, then $\exists \mathbf{v}.\ t < u \rightsquigarrow \mathbf{v}$ also does not hold. Indeed, if $\exists \mathbf{v}.\ t < u \rightsquigarrow \mathbf{v}$ holds then we have $u \longrightarrow^{\mathsf{h}} f(\mathbf{u}')$ and $\mathbf{t} < \mathbf{u}' \rightsquigarrow \mathbf{v}'$ for some $\mathbf{u}'$ and $\mathbf{v}'$. But from $f(\mathbf{u}) \equiv f(\mathbf{u}')$, confluence and the fact that both terms are head-normal, we get $\mathbf{u} \equiv \mathbf{u}'$, but Corollary 8.1 then implies $\exists \mathbf{v}.\ \mathbf{t} < \mathbf{u} \rightsquigarrow \mathbf{v}$. ■

Remark 8.5. The condition of strong normalization in Proposition 8.3 could be slightly weaken to require only weak normalization, for instance by exploring all the reducts in a fair manner until reaching the normal form, which would however be terribly inefficient.

Alternatively, we could ask for the existence of a *normalizing* strategy, that is, a strategy that eventually reaches a normal form when there exists one. For instance, for orthogonal systems, it is known that the *maximal-outermost* strategy is normalizing [vR97, vO99]. However, even if strategies derived from call-by-value have weaker theoretical guarantees, they are generally easier to implement efficiently — for instance, by using *normalization-by-evaluation*, as in the case of our implementation. Moreover, most theories used in practice are either strongly normalizing or not normalizing at all, and so it is questionable whether asking only for weak instead of strong normalization would bring any benefit in practice. □

8.3 Bidirectional syntax

In order to define the bidirectional type system, we first have to address the problem that some terms without annotations cannot be algorithmically typed. Indeed, suppose for instance that we want to type the term $@(\lambda(x.t), u)$ by inferring the sort of the principal argument of $@$ to recover A and B. But because $\lambda(x.t)$ is headed by a constructor it can only be bidirectionally typed in mode check, so we are stuck. One could think that this limitation is specific to bidirectional typing, however a famous result by Dowek shows that, in a dependently typed setting, the problem of typing non-annotated terms is actually undecidable in its full generality [Dow93]. Therefore, instead of defining the bidirectional system over the regular syntax of terms, we will define it over the *bidirectional syntax* which, given a signature Σ, is defined by the following grammar.

$$\begin{array}{llrll}
\boxed{\mathsf{tm}^{\mathsf{c}}\ \gamma} \ni & t, u, v ::= & | & c(\mathbf{t} \in \mathsf{msub}^{\mathsf{c}}\ \gamma\ \xi) & \text{if } c(\xi) \in \Sigma \\
 & & | & \underline{t} \in \mathsf{tm}^{\mathsf{i}}\ \gamma & \\
\boxed{\mathsf{tm}^{\mathsf{i}}\ \gamma} \ni & t, u, v ::= & | & x & \text{if } x \in \gamma \\
 & & | & d(t \in \mathsf{tm}^{\mathsf{i}}\ \gamma, \mathbf{t} \in \mathsf{msub}^{\mathsf{c}}\ \gamma\ \xi) & \text{if } d(\mathsf{x}, \xi) \in \Sigma \\
 & & | & t \in \mathsf{tm}^{\mathsf{c}}\ \gamma :: T \in \mathsf{tm}^{\mathsf{c}}\ \gamma & \\
\boxed{\mathsf{msub}^{\mathsf{c}}\ \gamma\ \xi} \ni & \mathbf{t}, \mathbf{u}, \mathbf{v} ::= & | & \varepsilon & \text{if } \xi = \cdot \\
 & & | & \mathbf{t} \in \mathsf{msub}^{\mathsf{c}}\ \gamma\ \xi', \vec{x}_\delta.t \in \mathsf{tm}^{\mathsf{c}}\ \gamma.\delta & \text{if } \xi = \xi', \mathsf{x}\{\delta\}
\end{array}$$

By separating between *checkable terms* $t \in \mathsf{tm}^{\mathsf{c}}\ \gamma$ and *inferrable terms* $t \in \mathsf{tm}^{\mathsf{i}}\ \gamma$ we are now able to specify that the principal argument of a destructor can only be an inferable term, avoiding the situation described in the previous paragraph. As a consequence of this, terms of the form $d(c(\mathbf{t}), \mathbf{u})$ are not directly part of the bidirectional syntax, and we must instead first turn $c(\mathbf{t})$ into an inferable term by adding a *(sort) ascription* $c(\mathbf{t}) :: T$, allowing us to then write $d(c(\mathbf{t}) :: T, \mathbf{u})$. We also have a symmetric operation of *embedding*, which creates a checkable term $\underline{t}$ from an inferable one t.

Example 8.2. The inferrable and checkable terms for the signature $\Sigma_{\lambda\Pi}$ are given respectively by the following grammars, where we omit the scope information.

$$t^{\mathsf{i}}, u^{\mathsf{i}} ::= x \mid t^{\mathsf{c}} :: T^{\mathsf{c}} \mid @(t^{\mathsf{i}}, u^{\mathsf{c}})$$
$$t^{\mathsf{c}}, u^{\mathsf{c}}, A^{\mathsf{c}}, B^{\mathsf{c}}, T^{\mathsf{c}} ::= \mathrm{Ty} \mid \mathrm{Tm}(A^{\mathsf{c}}) \mid \Pi(A^{\mathsf{c}}, x.B^{\mathsf{c}}) \mid \lambda(x.t^{\mathsf{c}}) \mid \underline{t^{\mathsf{i}}} \qquad \square$$

Because the bidirectional syntax requires us to add additional ascriptions when writing terms of the form $d(c(\mathbf{t}), \mathbf{u})$, one can wonder if this requirement might be too inconvenient in practice. If we remove sort ascriptions from the bidirectional syntax, then for most theories (like $\mathbb{T}_{\lambda\Pi}$) the checkable terms coincide exactly with the normal forms. As argued in other works [Nor07], users of type theory almost never write redexes, and because of this a large part of the bidirectional typing literature only supports the typing of normal forms [AC05, Nor07, AA11, Coq96, ACP11, AVW17], for which one needs no ascriptions. Our choice of *also* supporting ascriptions is simply a matter of giving users an extra convenience for the few situations in which writing a redex is more convenient, yet we expect that in most cases they will not be needed.

Given $t \in \mathsf{tm}^{\mathsf{c}}\ \gamma$ or $t \in \mathsf{tm}^{\mathsf{i}}\ \gamma$ we write $\ulcorner t \urcorner \in \mathsf{Tm}\ (\cdot)\ \gamma$ for its underlying term, obtained by forgetting the difference between checkable and inferable terms and by removing sort ascriptions. Similarly, if $\mathbf{t} \in \mathsf{msub}^{\mathsf{c}}\ \gamma\ \xi$ we write $\ulcorner \mathbf{t} \urcorner \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ for its underlying metavariable substitution.

Remark 8.6. Note that we have omitted metavariables from the bidirectional syntax. Even if metavariables are needed in the core syntax for *specifying* the theories in Section 7.2 (and the well-typed theories in Section 7.4), they are in general not needed for *using* them, and this is why they are in general omitted from most presentations of type theories. It is therefore reasonable to leave them out of the bidirectional syntax, as they would be of no utility for users. □

8.4 Bidirectional typing rules

Given a bidirectional theory $\mathbb{T}^{\flat}$, we can now define its bidirectional type system by the rules in Figure 8.3. The system is split in 4 judgments:

- $\Gamma \vdash T \Leftarrow \mathrm{sort}$: Checking that a checkable term T is a well-formed sort.
- $\Gamma \vdash t \Leftarrow T$: Checking that a checkable term t has sort T.
- $\Gamma \vdash t \Rightarrow T$: Inferring a sort T for an inferable term t.
- $\Gamma \mid \mathbf{v} : \Xi \vdash \mathbf{t} \Leftarrow \Theta$: Checking that a checkable metavariable substitution $\mathbf{t}$ can be typed by Θ, knowing that $\mathbf{v} : \Xi$.[2]

[2] The reader could argue that the most natural choice would be a judgment of the form $\Gamma \vdash \mathbf{t} \Leftarrow \Theta$. However, in some cases it will be necessary to check $\mathbf{v}.\ulcorner \mathbf{t} \urcorner : \Xi.\Theta$ knowing already that $\mathbf{v} : \Xi$ holds, so our more general judgment allows us to avoid reverifying that $\mathbf{v}$ is well-typed.

$$\boxed{\Gamma \vdash T \Leftarrow \mathsf{sort} \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|)}$$

$$c(\Xi)\ \mathsf{sort} \in \mathbb{T}^{\flat}\ \overset{\textsc{Sort}}{\frac{\Gamma \mid \varepsilon : (\cdot) \vdash \mathbf{t} \Leftarrow \Xi}{\Gamma \vdash c(\mathbf{t}) \Leftarrow \mathsf{sort}}}$$

$$\boxed{\Gamma \vdash t \Leftarrow T \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}\ (\cdot)\ |\Gamma|;\ t \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|)}$$

$$\begin{array}{c} c(\Xi_1\ ;\ \Xi_2\ ;\ \vec{t} \equiv \vec{u}) : T \in \mathbb{T}^{\flat} \\ \vec{t}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] \equiv \vec{u}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] \end{array}\ \overset{\textsc{Cons}}{\frac{T \prec T' \rightsquigarrow \mathbf{t}_1 \qquad \Gamma \mid \mathbf{t}_1 : \Xi_1 \vdash \mathbf{t}_2 \Leftarrow \Xi_2}{\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'}} \qquad T \equiv U\ \overset{\textsc{Switch}}{\frac{\Gamma \vdash t \Rightarrow T}{\Gamma \vdash \underline{t} \Leftarrow U}}$$

$$\boxed{\Gamma \vdash t \Rightarrow T \quad (\Gamma \in \mathsf{Ctx};\ T \in \mathsf{Tm}\ (\cdot)\ |\Gamma|;\ t \in \mathsf{tm}^{\mathsf{i}}\ |\Gamma|)}$$

$$x : T \in \Gamma\ \overset{\textsc{Var}}{\frac{}{\Gamma \vdash x \Rightarrow T}} \qquad \overset{\textsc{Ascr}}{\frac{\Gamma \vdash T \Leftarrow \mathsf{sort} \qquad \Gamma \vdash t \Leftarrow \ulcorner T \urcorner}{\Gamma \vdash t :: T \Rightarrow \ulcorner T \urcorner}}$$

$$d(\Xi_1\ ;\ \mathsf{x} :_{\mathsf{p}} T\ ;\ \Xi_2) : U \in \mathbb{T}^{\flat}\ \overset{\textsc{Dest}}{\frac{\begin{array}{c} \Gamma \vdash t \Rightarrow T' \qquad T \prec T' \rightsquigarrow \mathbf{t}_1 \\ \Gamma \mid \mathbf{t}_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{t}_2 \Leftarrow \Xi_2 \end{array}}{\Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U[\mathbf{t}_1, \ulcorner t \urcorner, \ulcorner \mathbf{t}_2 \urcorner]}}$$

$$\boxed{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi \quad (\Theta \in \mathsf{MCtx};\ \Gamma \in \mathsf{Ctx};\ \mathbf{v} \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ |\Theta|;\ \Xi \in \mathsf{MCtx}\ |\Theta|;\ \mathbf{t} \in \mathsf{msub}^{\mathsf{c}}\ |\Gamma|\ |\Xi|)}$$

$$\overset{\textsc{EmptyMSub}}{\frac{}{\Gamma \mid \mathbf{v} : \Theta \vdash \varepsilon \Leftarrow (\cdot)}} \qquad \overset{\textsc{ExtMSub}}{\frac{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi \qquad \Gamma.\Delta[\mathbf{v}, \ulcorner \mathbf{t} \urcorner] \vdash t \Leftarrow T[\mathbf{v}, \ulcorner \mathbf{t} \urcorner]}{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}, \vec{x}_{\Delta}.t \Leftarrow (\Xi, \mathsf{x}\{\Delta\} : T)}}$$

Figure 8.3: Bidirectional system defined by $\mathbb{T}^{\flat}$

As in the declarative type system of Figure 7.4, the most important rules are the ones that instantiate the schematic typing rules: Cons, Dest and Sort. However, differently from the declarative system, no more guessing is needed when building a type derivation. For instance, when using rule Dest with $d(t, \mathbf{u})$ the omitted arguments are no longer guessed, but instead recovered by inferring the sort of the principal argument t and then matching it against the associated pattern.

Example 8.1. Suppose we want to infer a sort for $@(t, u)$ in the theory $\mathbb{T}^{\flat}_{\lambda\Pi}$. To use rule Dest, we start by inferring a sort T' for t, and then we try to match it against the pattern $\mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}))$. If matching succeeds, we recover the arguments A and B, which together with t are then used in

$$\Gamma \mid (A, x.B, x.\ulcorner t \urcorner) : (\mathsf{A} : \mathrm{Ty},\ \mathsf{B}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Ty},\ \mathtt{t}\{x : \mathrm{Tm}(\mathsf{A})\} : \mathrm{Tm}(\mathsf{B})) \vdash (u) \Leftarrow (\mathsf{u} : \mathrm{Tm}(\mathsf{A}))$$

By applying the rules that define the judgment $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$, we see that this amounts to showing $\Gamma \vdash u \Leftarrow \mathrm{Tm}(A)$, and so the final shape of this "big-step derivation" is the following, which corresponds to the usual bidirectional rule for application.

$$\frac{\Gamma \vdash t \Rightarrow T' \qquad \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) < T' \rightsquigarrow A, x.B \qquad \Gamma \vdash u \Leftarrow \mathrm{Tm}(A)}{\Gamma \vdash @(t, u) \Rightarrow \mathrm{Tm}(B[\mathrm{id}_{\Gamma}, \ulcorner u \urcorner])}$$

□

8.5 Valid bidirectional theories

In order to establish the correctness of the bidirectional system with respect to the (declarative) typing rules of Figure 7.4, we first need to refine the notion of bidirectional theory with some extra assumptions. In the following, let us say that a pattern T is *destructor-free* if it contains no subterm headed by a destructor. Then, a bidirectional theory $\mathbb{T}^{\flat}$ is said to be *valid* when the following two conditions are met:

(A) Its underlying theory $\mathbb{T}$ is valid — recall that this means that **(I)** $\mathbb{T}$ is well-typed and its rewrite system satisfies **(II)** confluence and **(III)** subject reduction.

(B) For all patterns T, if $c(\Xi_1 \,;\, \Xi_2 \,;\, \vec{t} \equiv \vec{u}) : T \in \mathbb{T}^{\flat}$ or $d(\Xi_1 \,;\, \mathsf{x} :_{\mathsf{p}} T \,;\, \Xi_2) : U \in \mathbb{T}^{\flat}$, then T is rigid (with respect to the rewrite rules of $\mathbb{T}^{\flat}$) and destructor-free.

Assumption **(A)** is clearly reasonable, as we want the underlying theory of $\mathbb{T}^{\flat}$ to be well-behaved. For assumption **(B)**, the requirement that T is rigid will be needed when using Propositions 8.2 and 8.3 for showing completeness and decidability of the bidirectional system. Finally, the requirement that T is destructor-free is justified *a posteriori* by Lemma 8.3, needed for typing the recovered arguments and whose proof does not work when considering patterns with destructors.

Example 8.2. It is easy to see that the bidirectional theory $\mathbb{T}^{\flat}_{\lambda\Pi}$ is valid. Indeed, condition **(A)** follows from Example 7.7, and condition **(B)** can be straightforwardly verified. □

8.6 Correctness of bidirectional typing

In the following, when working over a bidirectional theory $\mathbb{T}^\flat$, all the declarative typing judgments are to be understood as in its underlying theory $\mathbb{T}$.

Soundness

In order to establish soundness, we will first need to know how to type the arguments recovered through matching. This is the role of the following Lemma 8.3, which roughly states that, given a well-typed pattern t such that the result of substituting $\mathbf{v}$ in t is also well-typed, one can conclude that $\mathbf{v}$ is also well-typed (though for Theorem 8.3 we need a slightly more general statement), an implication that generally does not hold when t is not a pattern. The proof of Lemma 8.3 in turn requires the following easy lemma.

Lemma 8.2 (Injectivity of rigid patterns). *Suppose that the rewrite system is confluent and let $t \in \mathsf{Tm}^{\mathsf{P}}\ \xi\ \delta$ be a rigid pattern. If for some $\mathbf{v}_1 \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ and $\mathbf{v}_2 \in \mathsf{MSub}\ (\cdot)\ \gamma\ \xi$ we have $t[\mathbf{v}_1] \equiv t[\mathbf{v}_2]$ then $\mathbf{v}_1 \equiv \mathbf{v}_2$.*

Proof. By confluence we have $t[\mathbf{v}_1] \longrightarrow^* u\ {}^*\!\!\longleftarrow t[\mathbf{v}_2]$ for some u. By iterating Lemma 8.1 we then get $u = t[\mathbf{v}'_1]$ with $\mathbf{v}_1 \longrightarrow^* \mathbf{v}'_1$ and $u = t[\mathbf{v}'_2]$ with $\mathbf{v}_2 \longrightarrow^* \mathbf{v}'_2$. Finally, from $t[\mathbf{v}'_1] = t[\mathbf{v}'_2]$ we can easily show $\mathbf{v}'_1 = \mathbf{v}'_2$ by induction on t, and so we get $\mathbf{v}_1 \equiv \mathbf{v}_2$. ∎

Lemma 8.3 (Substitution typing inversion for destructor-free patterns). *Suppose that* **(B)** *is satisfied and that the rewrite system is confluent. Let $\mathbf{v}_1 \in \mathsf{MSub}\ (\cdot)\ |\Delta|\ |\Theta_1|$ and $\mathbf{v}_2 \in \mathsf{MSub}\ (\cdot)\ |\Delta|\ |\Theta_2|$ and assume that one of the following three points is verified:*

- *$t \in \mathsf{Tm}^{\mathsf{P}}\ |\Theta_1|\ (\cdot)$ is destructor-free and $\Theta_1.\Theta_2 \vdash t : T$ and $\Delta \vdash t[\mathbf{v}_1] : T[\mathbf{v}_1, \mathbf{v}_2]$.*
- *$\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ |\Theta_1|\ (\cdot)\ |\Xi|$ is destructor-free and $\Theta_1.\Theta_2 \vdash \mathbf{t} : \Xi$ and $\Delta \vdash \mathbf{t}[\mathbf{v}_1] : \Xi$.*
- *$T \in \mathsf{Tm}^{\mathsf{P}}\ |\Theta_1|\ (\cdot)$ is destructor-free and $\Theta_1.\Theta_2 \vdash T$ sort and $\Delta \vdash T[\mathbf{v}_1]$ sort.*

Then we have $\Delta \vdash \mathbf{v} : \Theta_1$.

Proof. In order for the proof to go through, we need to show a stronger statement. Let $\mathbf{v} = \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ and $\Theta = \Theta_1.\Theta_2.\Theta_3$ with $\mathbf{v}_i \in \mathsf{MSub}\ (\cdot)\ |\Delta|\ |\Theta_i|$ for $i = 1, 2, 3$, and suppose moreover that $\Delta \vdash \mathbf{v}_1 : \Theta_1$ and that one of the following holds.

- $t \in \mathsf{Tm}^{\mathsf{P}}\ |\Theta_2|\ |\Gamma|$ is destructor-free and $\Theta; \Gamma \vdash t : T$ and $\Delta.\Gamma' \vdash t[\mathbf{v}_2] : T'$ with $\Gamma' \equiv \Gamma[\mathbf{v}]$ and $T' \equiv T[\mathbf{v}]$.
- $\mathbf{t}_2 \in \mathsf{MSub}^{\mathsf{P}}\ |\Theta_2|\ |\Gamma|\ |\Xi_2|$ is destructor-free and $\Theta; \Gamma \vdash \mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2$ and $\Delta.\Gamma' \vdash \mathbf{t}'_1, \mathbf{t}_2[\mathbf{v}_2] : \Xi_1.\Xi_2$ with $\Gamma' \equiv \Gamma[\mathbf{v}]$ and $\mathbf{t}'_1 \equiv \mathbf{t}_1[\mathbf{v}]$.
- $T \in \mathsf{Tm}^{\mathsf{P}}\ |\Theta_2|\ |\Gamma|$ is destructor-free and $\Theta; \Gamma \vdash T$ sort and $\Delta.\Gamma' \vdash T'[\mathbf{v}_2]$ sort with $\Gamma' \equiv \Gamma[\mathbf{v}]$.

Then we have $\Delta \vdash \mathbf{v}_1, \mathbf{v}_2 : \Theta_1.\Theta_2$. The proof is by induction on the pattern.

- Case $T = c(\mathbf{t} \in \mathsf{MSub}^{\mathsf{P}}\ |\Theta_2|\ |\Gamma|\ |\Xi|)$ for $c(\Xi)\ \mathsf{sort} \in \mathbb{T}^\flat$. By inversion on $\Theta; \Gamma \vdash T\ \mathsf{sort}$ and $\Delta.\Gamma' \vdash T'[\mathbf{v}_2]\ \mathsf{sort}$ we obtain $\Theta; \Gamma \vdash \mathbf{t} : \Xi$ and $\Delta.\Gamma' \vdash \mathbf{t}[\mathbf{v}_2] : \Xi$, so by i.h. we conclude.

- Case $t = c(\mathbf{t}_2 \in \mathsf{MSub}^{\mathsf{P}}\ |\Theta_2|\ |\Gamma|\ |\Xi_2|)$ for $c(\Xi_1\ ;\ \Xi_2\ ;\ \vec{t} \equiv \vec{u}) : U \in \mathbb{T}^\flat$. By inversion on $\Theta; \Gamma \vdash c(\mathbf{t}_2) : T$ and $\Delta.\Gamma' \vdash c(\mathbf{t}_2)[\mathbf{v}_2] : T'$ we obtain $\Theta; \Gamma \vdash \mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2$ with $T \equiv U[\mathbf{t}_1, \mathbf{t}_2]$ and $\Delta.\Gamma' \vdash \mathbf{t}'_1, \mathbf{t}_2[\mathbf{v}_2] : \Xi_1.\Xi_2$ with $T' \equiv U[\mathbf{t}'_1, \mathbf{t}_2[\mathbf{v}_2]]$, for some $\mathbf{t}_1$ and $\mathbf{t}'_1$. From **(B)** we get that $U \in \mathsf{Tm}^{\mathsf{P}}\ |\Xi_1|\ (\cdot)$ is rigid, and moreover we also have

$$U[\mathbf{t}'_1] = U[\mathbf{t}'_1, \mathbf{t}_2[\mathbf{v}_2]] \equiv T' \equiv T[\mathbf{v}] \equiv U[\mathbf{t}_1, \mathbf{t}_2][\mathbf{v}] \equiv U[\mathbf{t}_1[\mathbf{v}], \mathbf{t}_2[\mathbf{v}]] = U[\mathbf{t}_1[\mathbf{v}]]$$

  where the first and last equations follow from the fact that the metavariables in Ξ_2 do not occur in U. Therefore, by Lemma 8.2 we get $\mathbf{t}'_1 \equiv \mathbf{t}_1[\mathbf{v}]$, allowing us to apply the i.h. to conclude.

- Case $t = d(u, \mathbf{t})$. Impossible, because t is destructor-free.

- Case $t = \mathsf{x}\{\mathsf{id}_\Gamma\}$, in which case we must have $\Theta_2 = \mathsf{x}\{\Gamma_\mathsf{x}\} : T_\mathsf{x}$ for some Γ_x and T_x. By inversion on $\Theta; \Gamma \vdash t : T$ we get $T \equiv T_\mathsf{x}$ and $\Gamma \equiv \Gamma_\mathsf{x}$, and therefore $T' \equiv T_\mathsf{x}[\mathbf{v}]$ and $\Gamma' \equiv \Gamma_\mathsf{x}[\mathbf{v}]$. Moreover, as the only metavariables of Θ appearing in Γ_x and T_x are those of Θ_1, we have $\Gamma_\mathsf{x}[\mathbf{v}_1] = \Gamma_\mathsf{x}[\mathbf{v}]$ and $T_\mathsf{x}[\mathbf{v}_1] = T_\mathsf{x}[\mathbf{v}]$, and therefore $T' \equiv T_\mathsf{x}[\mathbf{v}_1]$ and $\Gamma' \equiv \Gamma_\mathsf{x}[\mathbf{v}_1]$. Then, because $\Theta \vdash$, we have $\Theta_1; \Gamma_\mathsf{x} \vdash T_\mathsf{x}\ \mathsf{sort}$, so by applying Proposition 7.6 with $\Delta \vdash \mathbf{v}_1 : \Theta_1$ we get $\Delta.\Gamma_\mathsf{x}[\mathbf{v}_1] \vdash T_\mathsf{x}[\mathbf{v}_1]\ \mathsf{sort}$. Now we can apply conversion and Proposition 7.7 to $\Delta.\Gamma' \vdash t[\mathbf{v}_2] : T'$ to get $\Delta.\Gamma_\mathsf{x}[\mathbf{v}_1] \vdash t[\mathbf{v}_2] : T_\mathsf{x}[\mathbf{v}_1]$. Because $t[\mathbf{v}_2] = \mathbf{v}_\mathsf{x}$ then together with $\Delta \vdash \mathbf{v}_1 : \Theta_1$ we can conclude $\Delta \vdash \mathbf{v}_1, \vec{x}_\Gamma.\mathbf{v}_\mathsf{x} : (\Theta_1, \mathsf{x}\{\Gamma_\mathsf{x}\} : T_\mathsf{x})$.

- Case $\mathbf{t}_2 = \varepsilon \in \mathsf{MSub}\ (\cdot)\ |\Gamma|\ (\cdot)$. Then the result follows by hypothesis.

- Case $\mathbf{t}_2 = \mathbf{u} \in \mathsf{MSub}\ |\Theta_{2l}|\ |\Gamma|\ |\Xi'_2|, \vec{x}.u \in \mathsf{Tm}^{\mathsf{P}}\ |\Theta_{2r}|\ |\Gamma|.|\Delta_\mathsf{x}|$ for $\Theta_2 = \Theta_{2l}.\Theta_{2r}$ and $\Xi_2 = \Xi'_2, \mathsf{x}\{\Delta_\mathsf{x}\} : T_\mathsf{x}$. Let $\mathbf{v}_2 = \mathbf{v}_{2l}, \mathbf{v}_{2l}$ be the splitting of $\mathbf{v}_2$ according to the decomposition $\Theta_2 = \Theta_{2l}.\Theta_{2r}$. By inversion on $\Theta; \Gamma \vdash \mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2$ and $\Delta.\Gamma' \vdash \mathbf{t}'_1, \mathbf{t}_2[\mathbf{v}_2] : \Xi_1.\Xi_2$ we obtain the following.

$$\frac{\begin{array}{c}\Theta; \Gamma \vdash \mathbf{t}_1, \mathbf{u} : \Xi_1.\Xi'_2 \\ \Theta; \Gamma.\Delta_\mathsf{x}[\mathbf{t}_1, \mathbf{u}] \vdash u : T_\mathsf{x}[\mathbf{t}_1, \mathbf{u}]\end{array}}{\Theta; \Gamma \vdash \mathbf{t}_1, \mathbf{u}, \vec{x}.u : (\Xi_1.\Xi'_2, \mathsf{x}\{\Delta_\mathsf{x}\} : T_\mathsf{x})}$$

$$\frac{\begin{array}{c}\Delta.\Gamma' \vdash \mathbf{t}'_1, \mathbf{u}[\mathbf{v}_{2l}] : \Xi_1.\Xi'_2 \\ \Delta.\Gamma'.\Delta_\mathsf{x}[\mathbf{t}'_1, \mathbf{u}[\mathbf{v}_{2l}]] \vdash u[\mathbf{v}_{2r}] : T_\mathsf{x}[\mathbf{t}'_1, \mathbf{u}[\mathbf{v}_{2l}]]\end{array}}{\Delta.\Gamma' \vdash \mathbf{t}'_1, \mathbf{u}[\mathbf{v}_{2l}], \vec{x}.u[\mathbf{v}_{2r}] : (\Xi_1.\Xi'_2, \mathsf{x}\{\Delta_\mathsf{x}\} : T_\mathsf{x})}$$

By the i.h. applied to the first premises we get $\Delta \vdash \mathbf{v}_1, \mathbf{v}_{2l} : \Theta_1.\Theta_{2l}$. Then, note that we have $(\Gamma.\Delta_{\mathsf{x}}[\mathbf{t}_1, \mathbf{u}])[\mathbf{v}] = \Gamma[\mathbf{v}].\Delta_{\mathsf{x}}[\mathbf{t}_1[\mathbf{v}], \mathbf{u}[\mathbf{v}_{2l}]] \equiv \Gamma'.\Delta_{\mathsf{x}}[\mathbf{t}'_1, \mathbf{u}[\mathbf{v}_{2l}]]$ and $T_{\mathsf{x}}[\mathbf{t}_1, \mathbf{u}][\mathbf{v}] \equiv T_{\mathsf{x}}[\mathbf{t}_1[\mathbf{v}], \mathbf{u}[\mathbf{v}_{2l}]] \equiv T_{\mathsf{x}}[\mathbf{t}'_1, \mathbf{u}[\mathbf{v}_{2l}]]$, so by the i.h. applied to the second premises we get $\Delta \vdash \mathbf{v}_1, \mathbf{v}_{2l}, \mathbf{v}_{2r} : \Theta_1.\Theta_{2l}.\Theta_{2r}$, concluding the proof. ■

We can now show soundness:

Theorem 8.3 (Soundness). *Suppose that $\mathbb{T}^\flat$ is valid.*

- *If* $\Gamma \vdash$ *and* $\Gamma \vdash t \Rightarrow T$ *then* $\Gamma \vdash \ulcorner t \urcorner : T$
- *If* $\Gamma \vdash T$ sort *and* $\Gamma \vdash t \Leftarrow T$ *then* $\Gamma \vdash \ulcorner t \urcorner : T$
- *If* $\Gamma \vdash$ *and* $\Gamma \vdash T \Leftarrow$ sort *then* $\Gamma \vdash \ulcorner T \urcorner$ sort
- *If* $\Gamma \vdash \mathbf{v} : \Theta$ *and* $\Theta.\Xi \vdash$ *and* $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$ *then* $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi$.

Proof. By induction on the derivation.

- Case Cons.

$$\begin{array}{c} c(\Xi_1 \,;\, \Xi_2 \,;\, \vec{t} \equiv \vec{u}) : T \in \mathbb{T}^\flat \\ \vec{t}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] \equiv \vec{u}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] \end{array} \dfrac{T \prec T' \rightsquigarrow \mathbf{t}_1 \qquad \Gamma \mid \mathbf{t}_1 : \Xi_1 \vdash \mathbf{t}_2 \Leftarrow \Xi_2}{\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'}$$

  By Proposition 8.1 we have $T' \longrightarrow^* T[\mathbf{t}_1]$, so because we have $\Gamma \vdash T'$ sort then by subject reduction we get $\Gamma \vdash T[\mathbf{t}_1]$ sort. We have $T \in \mathsf{Tm}^{\mathsf{P}}\ |\Xi_1|\ (\cdot)$, and well-typedness of $\mathbb{T}$ also gives $\Xi_1.\Xi_2 \vdash T$ sort, therefore by Lemma 8.3 we get $\Gamma \vdash \mathbf{t}_1 : \Xi_1$. We also have $\Xi_1.\Xi_2 \vdash$, therefore by applying the i.h. to the second premise we get $\Gamma \vdash \mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner : \Xi_1.\Xi_2$. Because we also have $\vec{t}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] \equiv \vec{u}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner]$ we can now derive $\Gamma \vdash c(\ulcorner \mathbf{t}_2 \urcorner) : T[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner]$, and because $T[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] = T[\mathbf{t}_1] \equiv T'$ and $\Gamma \vdash T'$ sort we can apply conversion to conclude $\Gamma \vdash c(\ulcorner \mathbf{u} \urcorner) : T'$.

- Case Switch.

$$T \equiv U \dfrac{\Gamma \vdash t \Rightarrow T}{\Gamma \vdash \underline{t} \Leftarrow U}$$

  By i.h. we have $\Gamma \vdash \ulcorner t \urcorner : T$, and because we have $\Gamma \vdash U$ sort and $T \equiv U$ we can apply the conversion rule to conclude $\Gamma \vdash \ulcorner \underline{t} \urcorner : U$.

- Case Dest.

$$d(\Xi_1 \,;\, \mathsf{x} :_{\mathsf{p}} T \,;\, \Xi_2) : U \in \mathbb{T}^\flat \ \dfrac{\begin{array}{c} \Gamma \vdash t \Rightarrow T' \qquad T \prec T' \rightsquigarrow \mathbf{t}_1 \\ \Gamma \mid \mathbf{t}_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{t}_2 \Leftarrow \Xi_2 \end{array}}{\Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U[\mathbf{t}_1, \ulcorner t \urcorner, \ulcorner \mathbf{t}_2 \urcorner]}$$

By i.h. we have $\Gamma \vdash \ulcorner t \urcorner : T'$. By Proposition 8.1 we have $T' \longrightarrow^* T[\mathbf{t}_1]$, so by Proposition 7.8 and subject reduction we get $\Gamma \vdash T[\mathbf{t}_1]$ sort. By well-typedness of $\mathbb{T}$, we have $\Xi_1.(\mathsf{x} : T).\Xi_2 \vdash U$ sort and therefore $\Xi_1 \vdash T$ sort, so because $T \in \mathsf{Tm}^{\mathsf{P}}\ |\Xi_1|\ (\cdot)$ we can apply Lemma 8.3 to derive $\Gamma \vdash \mathbf{t}_1 : \Xi_1$. From $\Gamma \vdash \ulcorner t \urcorner : T'$ and $T' \equiv T[\mathbf{t}_1]$ and $\Gamma \vdash T[\mathbf{t}_1]$ sort we can derive $\Gamma \vdash \ulcorner t \urcorner : T[\mathbf{t}_1]$, and thus $\Gamma \vdash \mathbf{t}_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T)$. We can now apply the i.h. to the third premise to derive $\Gamma \vdash \mathbf{t}_1, \ulcorner t \urcorner, \ulcorner \mathbf{t}_2 \urcorner : \Xi_1.(\mathsf{x} : T).\Xi_2$, and thus $\Gamma \vdash d(\ulcorner t \urcorner, \ulcorner \mathbf{t}_2 \urcorner) : U[\mathbf{t}_1, \ulcorner t \urcorner, \ulcorner \mathbf{t}_2 \urcorner]$.

- Case Var. Trivial.
- Case Ascr.

$$\frac{\Gamma \vdash T \Leftarrow \mathsf{sort} \qquad \Gamma \vdash t \Leftarrow \ulcorner T \urcorner}{\Gamma \vdash t :: T \Rightarrow \ulcorner T \urcorner}$$

By the i.h. applied to the first premise we have $\Gamma \vdash \ulcorner T \urcorner$ sort. Now we can apply the i.h. to the second premise and conclude $\Gamma \vdash \ulcorner t \urcorner : \ulcorner T \urcorner$, and because $\ulcorner t :: T \urcorner = \ulcorner t \urcorner$ we are done.

- Case Sort.

$$c(\Xi)\ \mathsf{sort} \in \mathbb{T}^{\flat}\ \frac{\Gamma \mid \varepsilon : (\cdot) \vdash \mathbf{t} \Leftarrow \Xi}{\Gamma \vdash c(\mathbf{t}) \Leftarrow \mathsf{sort}}$$

By well-typedness of $\mathbb{T}$ we have $\Xi \vdash$ and therefore we can apply the i.h. to show $\Gamma \vdash \ulcorner \mathbf{t} \urcorner : \Xi$, from which we conclude $\Gamma \vdash c(\ulcorner \mathbf{t} \urcorner)$ sort.

- Case EmptyMSub. Trivial.
- Case ExtMSub

$$\frac{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi \qquad \Gamma.\Delta[\mathbf{v}, \ulcorner \mathbf{t} \urcorner] \vdash t \Leftarrow T[\mathbf{v}, \ulcorner \mathbf{t} \urcorner]}{\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}, \vec{x}_\Delta.t \Leftarrow (\Xi, \mathsf{x}\{\Delta\} : T)}$$

By hypothesis we have $\Theta.\Xi, \mathsf{x}\{\Delta\} : T \vdash$, from which we get $\Theta.\Xi \vdash$ and $\Theta.\Xi; \Delta \vdash T$ sort. By the i.h. applied to the first premise we get $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi$, so by Proposition 7.6 applied with $\Theta.\Xi; \Delta \vdash T$ sort we get $\Gamma.\Delta[\mathbf{v}, \ulcorner \mathbf{t} \urcorner] \vdash T[\mathbf{v}, \ulcorner \mathbf{t} \urcorner]$ sort. Now we can apply the i.h. to the second premise and get $\Gamma.\Delta[\mathbf{v}, \ulcorner \mathbf{t} \urcorner] \vdash \ulcorner t \urcorner : T[\mathbf{v}, \ulcorner \mathbf{t} \urcorner]$, from which we can conclude $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner, \vec{x}.\ulcorner t \urcorner : \Theta.\Xi, \mathsf{x}\{\Delta\} : T$. ■

Annotability

We now want to show that the bidirectional system is complete with respect to the declarative typing rules, however what notion of completeness should we consider?

As argued by Dunfield and Krishnaswami [DK21], completeness in bidirectional typing should correspond to *annotability*: if $\Gamma \vdash t : T$ then for some t' with $\ulcorner t' \urcorner = t$ we should have $\Gamma \vdash t' \Leftarrow T$. In other words, t' should be equal to t modulo the insertion of sort ascriptions for when a destructor meets a constructor. Our proof of annotability will need the following lemma, ensuring that the bidirectional system respects conversion.

Lemma 8.4 (Bidirectional system respects conversion). *Suppose that $\mathbb{T}^\flat$ satisfies* **(B)** *and confluence.*

- *If $\Gamma \vdash t \Rightarrow T$ and $\Gamma' \equiv \Gamma$ then $\Gamma' \vdash t \Rightarrow T'$ for some $T' \equiv T$.*
- *If $\Gamma \vdash t \Leftarrow T$ and $\Gamma' \equiv \Gamma$ and $T' \equiv T$ then $\Gamma' \vdash t \Leftarrow T'$.*
- *If $\Gamma \vdash T \Leftarrow \mathsf{sort}$ and $\Gamma' \equiv \Gamma$ then $\Gamma' \vdash T \Leftarrow \mathsf{sort}$*
- *If $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$ and $\Gamma' \equiv \Gamma$ and $\mathbf{v}' \equiv \mathbf{v}$ then $\Gamma' \mid \mathbf{v}' : \Theta \vdash \mathbf{t} \Leftarrow \Xi$.*

Proof. By straightfoward induction, using Corollary 8.1 for cases Cons and Dest. ■

Our actual statement for annotability will be slightlier stronger than what we anticipated in the previous paragraphs. Let us call a bidirectional expression *minimal* if it contains no occurrences of $\underline{t :: T}$ or $\underline{t} :: T$. Our theorem will not only ensure that a regular term can be annotated into a bidirectional one, but also that the resulting term is minimal. In the end of the subsection, this will allow us to derive an alternative completeness result as a corollary of Theorem 8.4.

Theorem 8.4 (Annotability). *Suppose that $\mathbb{T}^\flat$ satisfies* **(B)** *and confluence.*

1. *If $\Gamma \vdash t : T$ then $\Gamma \vdash t' \Leftarrow T$ for some $t' \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|$ minimal with $\ulcorner t' \urcorner = t$.*
2. *If $\Gamma \vdash t : T$ then $\Gamma \vdash t' \Rightarrow T'$ for some $T' \equiv T$ and $t' \in \mathsf{tm}^{\mathsf{i}}\ |\Gamma|$ minimal with $\ulcorner t' \urcorner = t$.*
3. *If $\Gamma \vdash T\ \mathsf{sort}$ then $\Gamma \vdash T' \Leftarrow \mathsf{sort}$ for some $T' \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|$ minimal with $\ulcorner T' \urcorner = T$.*
4. *If $\Gamma \vdash \mathbf{v}, \mathbf{t} : \Theta.\Xi$ then $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}' \Leftarrow \Xi$ for some $\mathbf{t}' \in \mathsf{msub}^{\mathsf{c}}\ |\Gamma|\ |\Xi|$ minimal with $\ulcorner \mathbf{t}' \urcorner = \mathbf{t}$.*

Proof. The proof is by induction on the derivation. For the rules Var, TmSym, Conv we need to show both points 1 and 2, and so we will organize these cases accordingly.

- Case Var. Trivial.
- Case TmSym with a constructor rule $c(\Xi_1\ ;\ \Xi_2\ ;\ \vec{t} \equiv \vec{u}) : T \in \mathbb{T}^\flat$

  $$\vec{t}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1, \mathbf{t}_2]\ \frac{\Gamma \vdash \mathbf{t}_1, \mathbf{t}_2 : \Xi_1.\Xi_2 \qquad \Gamma \vdash T[\mathbf{t}_1, \mathbf{t}_2]\ \mathsf{sort}}{\Gamma \vdash c(\mathbf{t}_2) : T[\mathbf{t}_1, \mathbf{t}_2]}$$

  Fist note that, because $T \in \mathsf{Tm}^{\mathsf{P}}\ |\Xi_1|\ (\cdot)$ does not contain metavariables from Ξ_2, then $T[\mathbf{t}_1, \mathbf{t}_2] = T[\mathbf{t}_1]$, so the above occurrences of $T[\mathbf{t}_1, \mathbf{t}_2]$ can be replaced by $T[\mathbf{t}_1]$.

1 By Proposition 8.2 we have $T \prec T[\mathbf{t}_1] \rightsquigarrow \mathbf{t}'_1$ with $\mathbf{t}'_1 \equiv \mathbf{t}_1$. By the i.h. applied to the first premise, for some $\mathbf{t}'_2 \in \mathsf{msub}^{\mathsf{c}}\ |\Gamma|\ |\Xi_2|$ minimal we have $\Gamma \mid \mathbf{t}_1 : \Xi_1 \vdash \mathbf{t}'_2 \Leftarrow \Xi_2$ and $\ulcorner \mathbf{t}'_2 \urcorner = \mathbf{t}_2$, and by Lemma 8.4 with $\mathbf{t}_1 \equiv \mathbf{t}'_1$ we have $\Gamma \mid \mathbf{t}'_1 : \Xi_1 \vdash \mathbf{t}'_2 \Leftarrow \Xi_2$. Finally, we also have $\vec{t}[\mathbf{t}'_1, \ulcorner \mathbf{t}'_2 \urcorner] \equiv \vec{t}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}_1, \mathbf{t}_2] \equiv \vec{u}[\mathbf{t}'_1, \ulcorner \mathbf{t}'_2 \urcorner]$, so we conclude $\Gamma \vdash c(\mathbf{t}'_2) \Leftarrow T[\mathbf{t}_1]$ with $\ulcorner c(\mathbf{t}'_2) \urcorner = c(\mathbf{t}_2)$ and $c(\mathbf{t}'_2)$ minimal.

2 By the previous paragraph, we have $\Gamma \vdash t' \Leftarrow T[\mathbf{t}_1]$ for some $t' \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|$ minimal with $\ulcorner t' \urcorner = c(\mathbf{t}_2)$. Moreover, by the i.h. applied to the second premise we get $T' \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|$ such that $\Gamma \vdash T' \Leftarrow \mathsf{sort}$ and $\ulcorner T' \urcorner = T[\mathbf{t}_1]$, so we have $\Gamma \vdash t' :: T' \Rightarrow T[\mathbf{t}_1]$ with $\ulcorner t' :: T' \urcorner = c(\mathbf{t}_2)$. Finally, by inspection on the previous paragraph, t' is not of the form $\underline{t''}$, and so $t' :: T'$ is indeed minimal

- Case TmSym with a destructor rule $d(\Xi_1\ ;\ \mathsf{x} :_{\mathsf{p}} T\ ;\ \Xi_2) : U \in \mathbb{T}^\flat$.

$$\varepsilon \equiv \varepsilon\ \frac{\Gamma \vdash \mathbf{t}_1, t, \mathbf{t}_2 : \Xi_1.(\mathsf{x} : T).\Xi_2 \qquad \Gamma \vdash U[\mathbf{t}_1, t, \mathbf{t}_2]\ \mathsf{sort}}{\Gamma \vdash d(t, \mathbf{t}_2) : U[\mathbf{t}_1, t, \mathbf{t}_2]}$$

2 We can extract a strictly smaller derivation of $\Gamma \vdash t : T[\mathbf{t}_1]$, so by i.h. we get $t' \in \mathsf{tm}^{\mathsf{i}}\ |\Gamma|$ minimal with $\Gamma \vdash t' \Rightarrow T'$ and $T' \equiv T[\mathbf{t}_1]$ and $\ulcorner t' \urcorner = t$, and by Proposition 8.2 we get $T \prec T' \rightsquigarrow \mathbf{t}'_1$ with $\mathbf{t}'_1 \equiv \mathbf{t}_1$. By the i.h. again we also get $\mathbf{t}'_2 \in \mathsf{msub}^{\mathsf{c}}\ |\Gamma|\ |\Xi_2|$ minimal with $\Gamma \mid \mathbf{t}_1, \ulcorner t' \urcorner : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{t}'_2 \Leftarrow \Xi_2$ and $\ulcorner \mathbf{t}'_2 \urcorner = \mathbf{t}_2$, so by Lemma 8.4 with $\mathbf{t}_1, \ulcorner t' \urcorner \equiv \mathbf{t}'_1, \ulcorner t' \urcorner$ we have $\Gamma \mid \mathbf{t}'_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{t}'_2 \Leftarrow \Xi_2$. Finally, we conclude $\Gamma \vdash d(t', \mathbf{t}'_2) \Rightarrow U[\mathbf{t}'_1, \ulcorner t' \urcorner, \ulcorner \mathbf{t}'_2 \urcorner]$, and we indeed have $U[\mathbf{t}'_1, \ulcorner t' \urcorner, \ulcorner \mathbf{t}'_2 \urcorner] \equiv U[\mathbf{t}_1, t, \mathbf{t}_2]$ and $\ulcorner d(t', \mathbf{t}'_2) \urcorner = d(t, \mathbf{t}_2)$ and $d(t', \mathbf{t}'_2)$ minimal as required.

1 By the previous paragraph, for some $t' \in \mathsf{tm}^{\mathsf{i}}\ |\Gamma|$ minimal we have $\Gamma \vdash t' \Rightarrow U'$ with $U' \equiv U[\mathbf{t}_1, t, \mathbf{t}_2]$ and $\ulcorner t' \urcorner = d(t, \mathbf{t}_2)$. Therefore, by rule Switch we get $\Gamma \vdash \underline{t'} \Leftarrow U[\mathbf{t}_1, t, \mathbf{t}_2]$. Finally, by inspection on the previous paragraph, t' is not of the form $t'' :: T$ and so $\underline{t'}$ is indeed minimal.

- Case Conv.

$$T \equiv U\ \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U\ \mathsf{sort}}{\Gamma \vdash t : U}$$

1 By the i.h. applied to the first premise we get $t' \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|$ minimal with $\Gamma \vdash t' \Leftarrow T$ and $\ulcorner t' \urcorner = t$. By Lemma 8.4 with $T \equiv U$ we get $\Gamma \vdash t' \Leftarrow U$, so we are done.

2 By the i.h. applied to the first premise we get $t' \in \mathsf{tm}^{\mathsf{i}}\ |\Gamma|$ minimal with $\ulcorner t' \urcorner = t$ and $\Gamma \vdash t' \Rightarrow T'$ for some $T' \equiv T$. Because $T' \equiv T \equiv U$, we are done.

- Case SortSym with $c(\Xi)$ sort $\in \mathbb{T}^\flat$.

$$\frac{\Gamma \vdash \mathbf{t} : \Xi}{\Gamma \vdash c(\mathbf{t})\ \mathsf{sort}}$$

By i.h. we have $\mathbf{t}' \in \mathsf{msub}^{\mathsf{c}}\ |\Gamma|\ |\Xi|$ minimal such that $\Gamma \mid \varepsilon : (\cdot) \vdash \mathbf{t}' \Leftarrow \Xi$ and $\ulcorner \mathbf{t}' \urcorner = \mathbf{t}$, and thus $\Gamma \vdash c(\mathbf{t}') \Leftarrow \mathsf{sort}$ with $\ulcorner c(\mathbf{t}') \urcorner = c(\mathbf{t})$ and $c(\mathbf{t}')$ minimal.

- Cases EmptyMSub or ExtMSub with $\Gamma \vdash \mathbf{v}, \varepsilon : \Theta.(\cdot)$. Follows directly by applying the bidirectional rule EmptyMSub.

- Case ExtMSub with $\mathbf{t} = \mathbf{u}, \vec{x}.u$.

$$\frac{\Gamma \vdash \mathbf{v}, \mathbf{u} : \Theta.\Xi \qquad \Gamma.\Delta[\mathbf{v}, \mathbf{u}] \vdash u : T[\mathbf{v}, \mathbf{u}]}{\Gamma \vdash \mathbf{v}, \mathbf{u}, \vec{x}_{\Delta}.u : (\Theta.\Xi, \mathsf{x}\{\Delta\} : T)}$$

By the i.h. applied to both premises we get $\mathbf{u}' \in \mathsf{msub}^{\mathsf{c}}\ |\Gamma|\ |\Xi|$ minimal such that $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{u}' \Leftarrow \Xi$ and $\ulcorner \mathbf{u}' \urcorner = \mathbf{u}$, and we also get $u' \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma.\Delta[\mathbf{v}, \mathbf{u}]|$ minimal such that $\Gamma.\Delta[\mathbf{v}, \mathbf{u}] \vdash u' \Leftarrow T[\mathbf{v}, \mathbf{u}]$ and $\ulcorner u' \urcorner = u$. Therefore, we conclude $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{u}', \vec{x}.u' : (\Xi, \mathsf{x}\{\Delta\} : T)$ with $\ulcorner \mathbf{u}', \vec{x}.u' \urcorner = \mathbf{u}, \vec{x}.u$ and $\mathbf{u}', \vec{x}.u'$ minimal. ■

Ascription-free completeness

When considering a bidirectional system with ascriptions, completeness is nicely expressed by the notion of annotability. However, as mentioned in Section 8.3, some authors prefer to leave ascriptions out of the bidirectional syntax, given that they are generally only needed for writing redexes. In this setting, completeness instead ensures that, if a bidirectional term (seen as a regular one) is typable by the regular type system, than it is also typable by the bidirectional one.[3] We now show that, when considering the subset of the bidirectional syntax which removes ascriptions, this form of *ascription-free completeness* can be deduced almost for free from Theorem 8.4. More precisely, let us say that a bidirectional expression is *ascription-free* if it contains no occurrence of $t :: T$. We will then show that, if $\Gamma \vdash \ulcorner t \urcorner : T$ for t a checkable ascription-free term, then $\Gamma \vdash t \Leftarrow T$. Our main lemma for proving this will be the following one, stating that $\ulcorner - \urcorner$ satisfies a restricted form of injectivity.

Lemma 8.5 (Restricted injectivity of $\ulcorner - \urcorner$)**.**

- *If $t \in \mathsf{tm}^{\mathsf{c}}\ \gamma$ is ascription-free and $t' \in \mathsf{tm}^{\mathsf{c}}\ \gamma$ is minimal and $\ulcorner t \urcorner = \ulcorner t' \urcorner$ then $t = t'$.*

- *If $t \in \mathsf{tm}^{\mathsf{i}}\ \gamma$ is ascription-free and $t' \in \mathsf{tm}^{\mathsf{i}}\ \gamma$ is minimal and $\ulcorner t \urcorner = \ulcorner t' \urcorner$ then $t = t'$.*

- *If $\mathbf{t} \in \mathsf{msub}^{\mathsf{c}}\ \gamma\ \xi$ is ascription-free and $\mathbf{t}' \in \mathsf{msub}^{\mathsf{c}}\ \gamma\ \xi$ is minimal and $\ulcorner \mathbf{t} \urcorner = \ulcorner \mathbf{t}' \urcorner$ then $\mathbf{t} = \mathbf{t}'$.*

Proof. By straightfoward induction on t (or $\mathbf{t}$) and case analysis on t' (or $\mathbf{t}'$). ■

Ascription-free completeness now follows directly by composing Theorem 8.4 with Lemma 8.5.

[3] This is actually the notion of completeness we employed in our previous work [Fel24a].

Corollary 8.2 (Ascription-free completeness). *Suppose that $\mathbb{T}^\flat$ satisfies* **(B)** *and confluence.*

- *If $t \in \mathsf{tm}^{\mathsf{i}}\ |\Gamma|$ is ascription-free and $\Gamma \vdash \ulcorner t \urcorner : T$ then $\Gamma \vdash t \Rightarrow U$ with $T \equiv U$*
- *If $t \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|$ is ascription-free and $\Gamma \vdash \ulcorner t \urcorner : T$ then $\Gamma \vdash t \Leftarrow T$*
- *If $T \in \mathsf{tm}^{\mathsf{c}}\ |\Gamma|$ is ascription-free and $\Gamma \vdash \ulcorner T \urcorner\ \mathsf{sort}$ then $\Gamma \vdash T \Leftarrow \mathsf{sort}$*
- *If $\mathbf{t} \in \mathsf{msub}^{\mathsf{c}}\ |\Gamma|\ |\Xi|$ is ascription-free and $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi$ then $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$*

8.7 Decidability of bidirectional typing

We now come to the main property of interest of the bidirectional typing system: its decidability, allowing it to be used when implementing a type-checker for our framework. Our proof will need the following two lemmas, ensuring that matching and type-inference are *functional*, in the sense that when starting from convertible inputs we can only deduce convertible outputs.

Lemma 8.6 (Functionality of matching). *Suppose that the rewrite system is confluent.*

- *If $t \prec u \rightsquigarrow \mathbf{v}$ and $t \prec u' \rightsquigarrow \mathbf{v}'$ and $u \equiv u'$ then $\mathbf{v} \equiv \mathbf{v}'$*
- *If $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}$ and $\mathbf{t} \prec \mathbf{u}' \rightsquigarrow \mathbf{v}'$ and $\mathbf{u} \equiv \mathbf{u}'$ then $\mathbf{v} \equiv \mathbf{v}'$*

Proof. By induction on t or $\mathbf{t}$. The only interesting case is when $t = f(\mathbf{t})$, in which case we have $u \longrightarrow^{\mathsf{h}} f(\mathbf{u})$ and $\mathbf{t} \prec \mathbf{u} \rightsquigarrow \mathbf{v}$ and $u' \longrightarrow^{\mathsf{h}} f(\mathbf{u}')$ and $\mathbf{t} \prec \mathbf{u}' \rightsquigarrow \mathbf{v}'$. By applying confluence to $f(\mathbf{u}) \equiv f(\mathbf{u}')$, and using the fact the two terms are head normal, we get $\mathbf{u} \equiv \mathbf{u}'$, allowing us to apply the i.h. to conclude $\mathbf{v} \equiv \mathbf{v}'$. ■

Lemma 8.7 (Functionality of inference). *Suppose that the rewrite system is confluent. Then $\Gamma \equiv \Gamma'$ and $\Gamma \vdash t \Rightarrow T$ and $\Gamma' \vdash t \Rightarrow T'$ imply $T \equiv T'$*

Proof. By straightforward induction, using Lemma 8.6 for the case Dest. ■

A final hypothesis we will ask for ensuring the decidability of bidirectional typing is for the theory to be *strongly normalizing*, meaning that $\Theta; \Gamma \vdash t : T$ implies that t is s.n., and that $\Theta; \Gamma \vdash T\ \mathsf{sort}$ implies that T is s.n.[4] Indeed, type-checking with dependent types requires checking the conversion of terms (in rules Switch and Cons), whose decidability requires normalization. Moreover, strong normalization is also a requirement for the decidability of matching (Proposition 8.3).

Theorem 8.5 (Decidability of bidirectional typing). *Suppose that $\mathbb{T}^\flat$ is valid and s.n.*

- *If t is inferable and $\Gamma \vdash$ then the statement $\exists T.\ (\Gamma \vdash t \Rightarrow T)$ is decidable.*

[4] Note that both conditions are necessary, and one does not always imply the other

- *If t is checkable and $\Gamma \vdash T$* sort *then the statement $\Gamma \vdash t \Leftarrow T$ is decidable.*
- *If T is checkable and $\Gamma \vdash$ then the statement $\Gamma \vdash T \Leftarrow$* sort *is decidable.*
- *If $\mathbf{t}$ is checkable and $\Theta.\Xi \vdash$ and $\Gamma \vdash \mathbf{v} : \Theta$ then the statement $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi$ is decidable.*

Proof. By induction on the bidirectional expression.

- Case $c(\mathbf{t}_2)$ with $c(\Xi_1\ ;\ \Xi_2\ ;\ \vec{t} \equiv \vec{u}) : T \in \mathbb{T}^\flat$. Given Γ, T' with $\Gamma \vdash T'$ sort, we are to decide if $\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'$ holds. Because $\Gamma \vdash T'$ sort then T' is s.n., so together with **(B)** we can apply Proposition 8.3 to obtain that $\exists \mathbf{t}_1.\ T \prec T' \rightsquigarrow \mathbf{t}_1$ is decidable.
  - If $T \prec T' \rightsquigarrow \mathbf{t}_1$ does not hold for any $\mathbf{t}_1$, then $\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'$ is not derivable.
  - If $T \prec T' \rightsquigarrow \mathbf{t}_1$ holds, we apply Proposition 8.1 to derive $T' \longrightarrow^* T[\mathbf{t}_1]$. Then, by subject reduction applied to $\Gamma \vdash T'$ sort we get $\Gamma \vdash T[\mathbf{t}_1]$ sort, and by well-typedness of $\mathbb{T}$ we have $\Xi_1.\Xi_2 \vdash T$ sort, so by Lemma 8.3 we derive $\Gamma \vdash \mathbf{t}_1 : \Xi_1$. Because we have $\Xi_1.\Xi_2 \vdash$, by i.h. we get that $\Gamma \mid \mathbf{t}_1 : \Xi_1 \vdash \mathbf{t}_2 \Leftarrow \Xi_2$ is decidable.
    - If $\Gamma \mid \mathbf{t}_1 : \Xi_1 \vdash \mathbf{t}_2 \Leftarrow \Xi_2$ does not hold, if follows that $\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'$ does not hold. Indeed, if $\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'$ holds then we must have $T \prec T' \rightsquigarrow \mathbf{t}'_1$ and $\Gamma \mid \mathbf{t}'_1 : \Xi_1 \vdash \mathbf{t}_2 \Leftarrow \Xi_2$ for some $\mathbf{t}'_1$, so by Lemma 8.6 we get $\mathbf{t}_1 \equiv \mathbf{t}'_1$. But then Lemma 8.4 gives $\Gamma \mid \mathbf{t}_1 : \Xi_1 \vdash \mathbf{t}_2 \Leftarrow \Xi_2$, contradiction.
    - If $\Gamma \mid \mathbf{t}_1 : \Xi_1 \vdash \mathbf{t}_2 \Leftarrow \Xi_2$ holds then by Theorem 8.3 we get $\Gamma \vdash \mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner : \Xi_1.\Xi_2$. By well-typedness of $\mathbb{T}$ we have $\Xi_1.\Xi_2 \vdash \vec{t} : \Delta$ and $\Xi_1.\Xi_2 \vdash \vec{u} : \Delta$ for some Δ, so by Proposition 7.6 we get $\Gamma \vdash \mathrm{id}, \vec{t}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] : \Gamma.\Delta[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner]$ and $\Gamma \vdash \mathrm{id}, \vec{u}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] : \Gamma.\Delta[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner]$. Because the theory is s.n. and confluent, we can thus decide $\vec{t}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] \equiv \vec{u}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner]$. If this holds, then it follows that $\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'$ is derivable, otherwise it cannot be derivable. Indeed, if $\Gamma \vdash c(\mathbf{t}_2) \Leftarrow T'$ holds then for some $\mathbf{t}'_1$ we have $T \prec T' \rightsquigarrow \mathbf{t}'_1$ with $\vec{t}[\mathbf{t}'_1, \ulcorner \mathbf{t}_2 \urcorner] \equiv \vec{u}[\mathbf{t}'_1, \ulcorner \mathbf{t}_2 \urcorner]$, but Lemma 8.6 then implies $\mathbf{t}'_1 \equiv \mathbf{t}_1$ and thus $\vec{t}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner] \equiv \vec{u}[\mathbf{t}_1, \ulcorner \mathbf{t}_2 \urcorner]$, contradiction.
- Case $c(\mathbf{t})$ with $c(\Xi)$ sort $\in \mathbb{T}^\flat$. Given Γ with $\Gamma \vdash$ we are to decide if $\Gamma \vdash c(\mathbf{t}) \Leftarrow$ sort holds. By well-typedness of $\mathbb{T}$ we have $\Xi \vdash$, so by i.h. we get that $\Gamma \mid \varepsilon : (\cdot) \vdash \mathbf{t} \Leftarrow \Xi$ is decidable. Because this holds iff $\Gamma \vdash c(\mathbf{t}) \Leftarrow$ sort, it follows that the latter is also decidable.
- Case x. Trivial, as $\exists T.\ (\Gamma \vdash x \Rightarrow T)$ holds iff $x : T \in \Gamma$ for some T.
- Case $t :: T$. Given Γ with $\Gamma \vdash$ we are to decide if $\exists T'.\ (\Gamma \vdash t :: T \Rightarrow T')$ holds. By i.h., we can decide $\Gamma \vdash T \Leftarrow$ sort. If this does not hold, it follows that $\exists T'.\ (\Gamma \vdash t :: T \Rightarrow T')$ also does not hold. If $\Gamma \vdash T \Leftarrow$ sort holds, by Theorem 8.3 we get $\Gamma \vdash \ulcorner T \urcorner$ sort, hence by i.h. again we can decide $\Gamma \vdash t \Leftarrow \ulcorner T \urcorner$. If this is the case then we get $\Gamma \vdash t :: T \Rightarrow \ulcorner T \urcorner$, otherwise $\exists T'.\ (\Gamma \vdash t :: T \Rightarrow T')$ does not hold.

- Case $d(t, \mathbf{t}_2)$ with $d(\Xi_1 \,;\, \mathsf{x} :_{\mathrm{p}} T \,;\, \Xi_2) : U \in \mathbb{T}^\flat$. Given Γ with $\Gamma \vdash$ we are to decide if $\exists U'.\ (\Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U')$ holds. By i.h. it follows that $\exists T'.\ \Gamma \vdash t \Rightarrow T'$ is decidable.
  - If $\exists T'\ .(\Gamma \vdash t \Rightarrow T')$ does not hold, it is clear that $\exists U'.\ \Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U'$ does not hold.
  - If $\Gamma \vdash t \Rightarrow T'$ is derivable, then by Theorem 8.3 it follows that $\Gamma \vdash \ulcorner t \urcorner : T'$ holds. Therefore, T' is s.n. and we have **(B)** by hypothesis, so by Proposition 8.3 it follows that $\exists \mathbf{t}_1.\ T \prec T' \rightsquigarrow \mathbf{t}_1$ is decidable.
    - If $T \prec T' \rightsquigarrow \mathbf{t}_1$ holds for no $\mathbf{t}_1$, it follows that $\exists U'.\ \Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U'$ does not hold neither. Indeed, if $\Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U'$ holds for some U', then we have $\Gamma \vdash t \Rightarrow T''$ and $T \prec T'' \rightsquigarrow \mathbf{t}'_1$ for some T'' and $\mathbf{t}'_1$. But by Lemma 8.7 and Corollary 8.1 we get $T \prec T' \rightsquigarrow \mathbf{t}_1$ for some $\mathbf{t}_1$, contradiction.
    - If $T \prec T' \rightsquigarrow \mathbf{t}_1$ is derivable, then by Proposition 8.1 we get $T' \longrightarrow^* T[\mathbf{t}_1]$, so by subject reduction applied to $\Gamma \vdash T'$ sort we get $\Gamma \vdash T[\mathbf{t}_1]$ sort. By well-typedness of $\mathbb{T}$ we have $\Xi_1.(\mathsf{x} : T).\Xi_2 \vdash U$ sort and thus $\Xi_1 \vdash T$ sort. Therefore by Lemma 8.3 we get $\Gamma \vdash \mathbf{t}_1 : \Xi_1$ and so $\Gamma \vdash \mathbf{t}_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T)$. By i.h., the statement $\Gamma \mid \mathbf{t}_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{t}_2 \Leftarrow \Xi_2$ is decidable. If it holds, we conclude that $\Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U[\mathbf{t}_1, \ulcorner t \urcorner, \ulcorner \mathbf{t}_2 \urcorner]$ also holds. Otherwise $\Gamma \vdash d(t, \mathbf{t}_2) \Rightarrow U'$ cannot hold for no U'. Indeed, this would imply $\Gamma \vdash t \Rightarrow T''$ and $T \prec T'' \rightsquigarrow \mathbf{t}'_1$ and $\Gamma \mid \mathbf{t}'_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{t}_2 \Leftarrow \Xi_2$ for some T'' and $\mathbf{t}'_1$, so Lemmas 8.4 and 8.7 and Corollary 8.1 would give $\Gamma \mid \mathbf{t}_1, \ulcorner t \urcorner : (\Xi_1, \mathsf{x} : T) \vdash \mathbf{t}_2 \Leftarrow \Xi_2$, a contradiction.

- Case $\underline{u}$. Given Γ, T with $\Gamma \vdash T$ sort, we are to decide if $\Gamma \vdash \underline{u} \Leftarrow T$ holds. By i.h. we have that $\exists U.\ \Gamma \vdash u \Rightarrow U$ is decidable. If this statement does not hold, it follows that $\Gamma \vdash \underline{u} \Leftarrow T$ does not hold. If $\exists U.\ \Gamma \vdash u \Rightarrow U$ holds, then by Theorem 8.3 we get $\Gamma \vdash \ulcorner u \urcorner : U$, which by Proposition 7.8 implies $\Gamma \vdash U$ sort. We also have $\Gamma \vdash T$ sort so it follows that both U and T are s.n., allowing us to decide $T \equiv U$. If this is the case, then it follows that $\Gamma \vdash \underline{u} \Leftarrow T$ holds. Otherwise $\Gamma \vdash \underline{u} \Leftarrow T$ cannot hold, as this would imply $\Gamma \vdash u \Rightarrow U'$ for some $U' \equiv T$, but then Lemma 8.7 implies $U \equiv U'$ and so $T \equiv U$, contradiction.

- Case ε. Trivial, as $\Gamma \mid \mathbf{v} : \Theta \vdash \varepsilon \Leftarrow \Xi$ holds iff $\Xi = \cdot$.

- Case $\mathbf{t}, \vec{x}.t$. Given $\Gamma, \Theta, \Xi, \mathbf{v}$ with $\Theta.\Xi \vdash$ and $\Gamma \vdash \mathbf{v} : \Theta$, we are to decide if $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}, \vec{x}.t \Leftarrow \Xi$ holds. If $\Xi = \cdot$ then this clearly does not hold, so let us now suppose that $\Xi = \Xi', \mathsf{x}\{\Delta\} : T$. From $\Theta.\Xi', \mathsf{x}\{\Delta\} : T \vdash$ we then get $\Theta.\Xi' \vdash$ and $\Theta.\Xi'; \Delta \vdash T$ sort. By i.h. we then get that $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi'$ is decidable. If this does not hold, then it is clear that $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}, \vec{x}.t \Leftarrow \Xi$ is not derivable, so in the following we assume that we have $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t} \Leftarrow \Xi'$. Then, by Theorem 8.3 we get $\Gamma \vdash \mathbf{v}, \ulcorner \mathbf{t} \urcorner : \Theta.\Xi'$, so by applying Proposition 7.6 with $\Theta.\Xi'; \Delta \vdash T$ sort we get $\Gamma.\Delta[\mathbf{v}, \ulcorner \mathbf{t} \urcorner] \vdash T[\mathbf{v}, \ulcorner \mathbf{t} \urcorner]$ sort.

By i.h. we therefore get that $\Gamma.\Delta[\mathbf{v}, \ulcorner\mathbf{t}\urcorner] \vdash t \Leftarrow T[\mathbf{v}, \ulcorner\mathbf{t}\urcorner]$ is decidable, hence by testing this statement we can decide $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{t}, \vec{x}.t \Leftarrow \Xi', \mathsf{x}\{\Delta\} : T$. ■

Chapter 9

A Zoo of Bidirectional Theories

In the previous chapters we have illustrated our framework with the theory $\mathbb{T}_{\lambda\Pi}$ and its bidirectional counterpart $\mathbb{T}^{\flat}_{\lambda\Pi}$. We now showcase the generality of our framework by going through various other examples of bidirectional theories we support.

All of the bidirectional theories we present in this chapter are valid:

- Condition **(A)**, stating that the underlying theory is valid, follows because:
  - Condition **(I)**, stating that schematic rules are well-typed, can be verified either manually or using the implementation.
  - Condition **(II)**, stating that rewrite rules are confluent, follows directly from Mayr and Nipkow's orthogonality criterion [MN98] for almost all examples, and for the other ones we discuss explicitly how confluence can be shown.
  - Condition **(III)**, stating that rewrite rules satisfy subject reduction, follows by applying Proposition 7.10 and verifying manually that all rewrite rules preserve typing, using the same strategy as in Example 7.8.
- Condition **(B)**, stating that the patterns of constructor and destructor rules are rigid and destructor-free, can also be readily verified manually or using the implementation.

Throughout this chapter, we use the informal notation for schematic rules discussed in Sections 7.2 and 8.1 for readability purposes. For more information about the examples we refer to the files in the implementation in which they are detailed.

9.1 Inductive types

Our framework supports the definition of arbitrary *inductive types* [ML84, PM93]. For instance, starting from $\mathbb{T}^{\flat}_{\lambda\Pi}$, dependent sums can be defined by the following declarations. Note that, as one would wish, the parameters A and B are completely omitted in the constructor and the projections.

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty}}{\Sigma(\mathtt{A}, x.\mathtt{B}\{x\}) : \mathrm{Ty}} \qquad \frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty} \\ \mathtt{t} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{u} : \mathrm{Tm}(\mathtt{B}\{\mathtt{t}\})\end{array}}{\mathrm{pair}(\mathtt{t}, \mathtt{u}) : \mathrm{Tm}(\Sigma(\mathtt{A}, x.\mathtt{B}\{x\}))}$$

$$\frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty} \\ \mathtt{t} :_{\mathrm{p}} \mathrm{Tm}(\Sigma(\mathtt{A}, x.\mathtt{B}\{x\}))\end{array}}{\mathrm{proj}_1(\mathtt{t}) : \mathrm{Tm}(\mathtt{A})} \qquad \frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty} \\ \mathtt{t} :_{\mathrm{p}} \mathrm{Tm}(\Sigma(\mathtt{A}, x.\mathtt{B}\{x\}))\end{array}}{\mathrm{proj}_2(\mathtt{t}) : \mathrm{Tm}(\mathtt{B}\{\mathrm{proj}_1(\mathtt{t})\})}$$

$$\mathrm{proj}_1(\mathrm{pair}(\mathtt{t}, \mathtt{u})) \longmapsto \mathtt{t} \qquad \mathrm{proj}_2(\mathrm{pair}(\mathtt{t}, \mathtt{u})) \longmapsto \mathtt{u}$$

Dependent sums are an example of *negative types*, which are types that are eliminated by means of projections — these correspond to *record types* in proof assistants like Agda. We can also define *positive types*,[1] which feature instead a dependent eliminator. The main example of positive inductive type are W types, which can be used to define any other positive inductive type [Hug21]. Once again, note how the parameters $\mathtt{A}$ and $\mathtt{B}$ are omitted from both the constructor sup and the eliminator $\mathrm{rec}_{\mathrm{W}}$.

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty}}{\mathrm{W}(\mathtt{A}, x.\mathtt{B}\{x\}) : \mathrm{Ty}} \qquad \frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty} \\ \mathtt{a} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{f} : \mathrm{Tm}(\Pi(\mathtt{B}\{\mathtt{a}\}, \_.\mathrm{W}(\mathtt{A}, x.\mathtt{B}\{x\})))\end{array}}{\mathrm{sup}(\mathtt{a}, \mathtt{f}) : \mathrm{Tm}(\mathrm{W}(\mathtt{A}, x.\mathtt{B}\{x\}))}$$

$$\frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty} \qquad \mathtt{t} :_{\mathrm{p}} \mathrm{Tm}(\mathrm{W}(\mathtt{A}, x.\mathtt{B}\{x\})) \qquad x : \mathrm{Tm}(\mathrm{W}(\mathtt{A}, x.\mathtt{B}\{x\})) \vdash \mathtt{P} : \mathrm{Ty} \\ x : \mathrm{Tm}(\mathtt{A}), y : \mathrm{Tm}(\Pi(\mathtt{B}\{x\}, \_.\mathrm{W}(\mathtt{A}, x.\mathtt{B}\{x\}))), z : \mathrm{Tm}(\Pi(\mathtt{B}\{x\}, x'.\mathtt{P}\{@(y, x')\})) \vdash \mathtt{p} : \mathrm{Tm}(\mathtt{P}\{\mathrm{sup}(x, y)\})\end{array}}{\mathrm{rec}_{\mathrm{W}}(\mathtt{t}, x.\mathtt{P}\{x\}, xyz.\mathtt{p}\{x, y, z\}) : \mathrm{Tm}(\mathtt{P}\{\mathtt{t}\})}$$

$$\mathrm{rec}_{\mathrm{W}}(\mathrm{sup}(\mathtt{a}, \mathtt{f}), x.\mathtt{P}\{x\}, xyz.\mathtt{p}\{x, y, z\}) \longmapsto \mathtt{p}\{\mathtt{a}, \mathtt{f}, \lambda(x'.\mathrm{rec}_{\mathrm{W}}(@(\mathtt{f}, x'), x.\mathtt{P}\{x\}, xyz.\mathtt{p}\{x, y, z\}))\}$$

9.2 Indexed inductive types

The types of the previous subsection are non-indexed, in the sense that they are specified uniformly in their parameters (in the examples, $\mathtt{A}$ and $\mathtt{B}$). Our framework however also supports *indexed* inductive types [Dyb94, PM93], which are also specified by indices that can vary along the definition. An example of such a type is the one of vectors, for which the length $\mathtt{n}$ takes the value 0 in the constructor nil but $\mathrm{S}(\mathtt{m})$ in the constructor cons.

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{n} : \mathrm{Tm}(\mathrm{Nat})}{\mathrm{Vec}(\mathtt{A}, \mathtt{n}) : \mathrm{Ty}} \qquad \frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{n} : \mathrm{Tm}(\mathrm{Nat}) \qquad \mathtt{n} \equiv 0}{\mathrm{nil} : \mathrm{Tm}(\mathrm{Vec}(\mathtt{A}, \mathtt{n}))}$$

$$\frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad \mathtt{n} : \mathrm{Tm}(\mathrm{Nat}) \qquad \mathtt{m} : \mathrm{Tm}(\mathrm{Nat}) \qquad \mathtt{t} : \mathrm{Tm}(\mathtt{A}) \\ \mathtt{l} : \mathrm{Tm}(\mathrm{Vec}(\mathtt{A}, \mathtt{m})) \qquad \mathtt{n} \equiv \mathrm{S}(\mathtt{m})\end{array}}{\mathrm{cons}(\mathtt{m}, \mathtt{t}, \mathtt{l}) : \mathrm{Tm}(\mathrm{Vec}(\mathtt{A}, \mathtt{n}))}$$

[1] Note that this terminology is unrelated to the *strict-positivity* condition [PM93]. In this thesis, all considered inductive types are strictly positive.

$$\frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad \mathtt{n} : \mathrm{Tm}(\mathrm{Nat}) \qquad \mathtt{l} :_{\mathrm{p}} \mathrm{Tm}(\mathrm{Vec}(\mathtt{A}, \mathtt{n})) \\ x : \mathrm{Tm}(\mathrm{Nat}), y : \mathrm{Tm}(\mathrm{Vec}(\mathtt{A}, x)) \vdash \mathtt{P} : \mathrm{Ty} \qquad \mathtt{pnil} : \mathrm{Tm}(\mathtt{P}\{0, \mathrm{nil}\}) \\ x : \mathrm{Tm}(\mathrm{Nat}), y : \mathrm{Tm}(\mathtt{A}), z : \mathrm{Tm}(\mathrm{Vec}(\mathtt{A}, x)), w : \mathrm{Tm}(\mathtt{P}\{x, z\}) \vdash \mathtt{pcons} : \mathrm{Tm}(\mathtt{P}\{\mathrm{S}(x), \mathrm{cons}(x, y, z)\})\end{array}}{\mathrm{rec}_{\mathrm{Vec}}(\mathtt{l}, x.\mathtt{P}\{x\}, \mathtt{pnil}, xyzw.\mathtt{pcons}\{x, y, z, w\}) : \mathrm{Tm}(\mathtt{P}\{\mathtt{n}, \mathtt{l}\})}$$

As anticipated in Section 8.1, in order for the above definition to be a bidirectional theory in the sense of Section 8.1, we have applied what is known as the *fording* technique,[2] consisting in indexing constructors over a fresh metavariable $\mathtt{n}$, which is then constrained by equational premises to be equal to the actual indices (0 and $\mathrm{S}(\mathtt{m})$ in the above example).

While applying fording for nil is not essential, it is in the case of cons. This becomes visible when writing the rewrite rules for $\mathrm{rec}_{\mathrm{Vec}}$: without the use of fording, the argument $\mathtt{m}$ of cons would be omitted, yet if $\mathtt{m}$ were not available in the left-hand side of the second rule, how would we fill $\square$ in $\mathtt{pcons}\{\square, \mathtt{t}, \mathtt{l}, \mathrm{rec}_{\mathrm{Vec}}(\ldots)\}$?

$$\begin{array}{l}\mathrm{rec}_{\mathrm{Vec}}(\mathrm{nil}, xy.\mathtt{P}\{x, y\}, \mathtt{pnil}, xyzw.\mathtt{pcons}\{x, y, z, w\}) \longmapsto \mathtt{pnil} \\ \mathrm{rec}_{\mathrm{Vec}}(\mathrm{cons}(\mathtt{m}, \mathtt{t}, \mathtt{l}), xy.\mathtt{P}\{x, y\}, \mathtt{pnil}, xyzw.\mathtt{pcons}\{x, y, z, w\}) \longmapsto \\ \qquad \mathtt{pcons}\{\mathtt{m}, \mathtt{t}, \mathtt{l}, \mathrm{rec}_{\mathrm{Vec}}(\mathtt{l}, xy.\mathtt{P}\{x, y\}, \mathtt{pnil}, xyzw.\mathtt{pcons}\{x, y, z, w\})\}\end{array}$$

Another example of indexed type is equality, also known as Martin-Löf's *identity type*:

$$\frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{a} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{b} : \mathrm{Tm}(\mathtt{A})}{\mathrm{Eq}(\mathtt{A}, \mathtt{a}, \mathtt{b}) : \mathrm{Ty}} \qquad \frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{a} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{b} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{a} \equiv \mathtt{b}}{\mathrm{refl} : \mathrm{Tm}(\mathrm{Eq}(\mathtt{A}, \mathtt{a}, \mathtt{b}))}$$

$$\frac{\begin{array}{c}\mathtt{A} : \mathrm{Ty} \qquad \mathtt{a} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{b} : \mathrm{Tm}(\mathtt{A}) \qquad \mathtt{t} :_{\mathrm{p}} \mathrm{Tm}(\mathrm{Eq}(\mathtt{A}, \mathtt{a}, \mathtt{b})) \\ x : \mathrm{Tm}(\mathtt{A}), y : \mathrm{Tm}(\mathrm{Eq}(\mathtt{A}, \mathtt{a}, x)) \vdash \mathtt{P} : \mathrm{Ty} \qquad \mathtt{p} : \mathrm{Tm}(\mathtt{P}\{\mathtt{a}, \mathrm{refl}\})\end{array}}{\mathrm{J}(\mathtt{t}, xy.\mathtt{P}\{x, y\}, \mathtt{p}) : \mathrm{Tm}(\mathtt{P}\{\mathtt{b}, \mathtt{t}\})} \qquad \mathrm{J}(\mathrm{refl}, xy.\mathtt{P}\{x, y\}, \mathtt{p}) \longmapsto \mathtt{p}$$

Here, we use Paulin-Mohring's eliminator [PM93], which allows for refl to carry no arguments whatsoever. It is also possible to use Martin-Löf original eliminator [ML75] — which is known to be equivalent to Paulin-Mohring's eliminator [Str93, Addendum] — at the cost of adding an extra annotation to refl.

9.3 Higher-order logic

It is well-known that the Curry-Howard correspondence allows us to embed many kinds of logic into type theories, enabling us to use the previously introduced types as propositions. Nevertheless, it can be useful sometimes to explicitly separate types from propositions, as done for instance in the proof assistant Coq. Let us illustrate how this can be done in our

[2] Fording was initially introduced by Coquand for being used in the Agda proof assistant — see the discussion by Dybjer and Setzer in [DS06, Subsection 1.4]. The name fording is however due to McBride, who rediscovered this technique in the context of his work on compiling dependent pattern matching to eliminators [McB00].

framework by defining a variant of Higher-Order Logic. We start by extending $\mathbb{T}^{\flat}_{\lambda\Pi}$ with a type of propositions and a sort $\mathrm{Prf}(\mathsf{P})$ for representing the judgment form "□ is a proof of P", where □ stands for the subject.

$$\frac{}{\mathrm{Prop} : \mathrm{Ty}} \qquad \frac{\mathsf{P} : \mathrm{Tm}(\mathrm{Prop})}{\mathrm{Prf}(\mathsf{P})\ \mathsf{sort}}$$

We can then add arbitrary connectives or quantifiers. For instance, we can add universe quantification with the following declarations. We refer to the file `hol.bitts` of the implementation in which we also add implication and define conjunction using the *impredicative encoding* [GTL89].

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop})}{\forall(\mathsf{A}, x.\mathsf{P}\{x\}) : \mathrm{Tm}(\mathrm{Prop})} \qquad \frac{\begin{array}{c}\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \\ x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{p} : \mathrm{Prf}(\mathsf{P})\end{array}}{\forall_{\mathrm{i}}(x.\mathsf{p}\{x\}) : \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\}))}$$

$$\frac{\begin{array}{c}\mathsf{A} : \mathrm{Ty} \qquad x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{P} : \mathrm{Tm}(\mathrm{Prop}) \\ \mathsf{q} :_{\mathrm{p}} \mathrm{Prf}(\forall(\mathsf{A}, x.\mathsf{P}\{x\})) \qquad \mathsf{t} : \mathrm{Tm}(\mathsf{A})\end{array}}{\forall_{\mathrm{e}}(\mathsf{q}, \mathsf{t}) : \mathrm{Prf}(\mathsf{P}\{\mathsf{t}\})} \qquad \forall_{\mathrm{e}}(\forall_{\mathrm{i}}(x.\mathsf{p}\{x\}), \mathsf{t}) \longmapsto \mathsf{p}\{\mathsf{t}\}$$

9.4 Universes

In dependent type theories, types can be reified as terms by adding *universes*. Starting from $\mathbb{T}^{\flat}_{\lambda\Pi}$, we define a *Tarski-style universe* by adding a type U of *codes* and a decoding function El mapping each code to an associated type. We then must close U under the type formers of our theory, by adding the codes u for U and π for Π, and stating that El decodes them to the expected types.

$$\frac{}{\mathrm{U} : \mathrm{Ty}} \qquad \frac{\mathsf{a} :_{\mathrm{p}} \mathrm{Tm}(\mathrm{U})}{\mathrm{El}(\mathsf{a}) : \mathrm{Ty}} \qquad \frac{}{\mathrm{u} : \mathrm{Tm}(\mathrm{U})} \qquad \mathrm{El}(\mathrm{u}) \longmapsto \mathrm{U}$$

$$\frac{\mathsf{a} : \mathrm{Tm}(\mathrm{U}) \qquad x : \mathrm{Tm}(\mathrm{El}(\mathsf{a})) \vdash \mathsf{b} : \mathrm{Tm}(\mathrm{U})}{\pi(\mathsf{a}, x.\mathsf{b}\{x\}) : \mathrm{Tm}(\mathrm{U})} \qquad \mathrm{El}(\pi(\mathsf{a}, x.\mathsf{b}\{x\})) \longmapsto \Pi(\mathrm{El}(\mathsf{a}), x.\mathrm{El}(\mathsf{b}\{x\}))$$

For illustrative purposes, in the above we have defined a *type-in-type* universe, which is known to be inconsistent [Coq86]. This can however be easily solved if we stratify universes into an hierarchy, by instead introducing a family of symbols $\mathrm{U}_l, \mathrm{El}_l, \ldots$ indexed by some set $l \in \mathscr{L}$ of *universe levels*. In particular, this allows us to define a Tarski-style variant of Pure Type Systems, which are usually presented using *Russell-style* universes.

Internal universe levels

By indexing the above symbols externally, they become annotated with levels l in their names, as in Chapter 5. An alternative approach is to index them *internally*, which then allows us to leverage our support for erased arguments. Let us illustrate this by taking $\mathscr{L} := \mathbb{N}$. We start by declaring a sort of levels along with constructors for zero and successor.

$$\frac{}{\mathrm{Lvl}\ \mathsf{sort}} \qquad\qquad \frac{}{\mathtt{0} : \mathrm{Lvl}} \qquad\qquad \frac{\mathtt{l} : \mathrm{Lvl}}{\mathrm{S}(\mathtt{l}) : \mathrm{Lvl}}$$

We then update the definitions of $\mathrm{U}, \mathrm{El}, \mathrm{u}, \pi$ in the following manner, so that they now take a level as an argument. With these definitions, we can omit the level annotations in El and π. Note that in the case of u we cannot omit $\mathtt{l}$ from the syntax, otherwise we would not be able to define the rewrite rule $\mathrm{El}(\mathrm{u}) \longmapsto \mathrm{U}(?)$.

$$\frac{\mathtt{l} : \mathrm{Lvl}}{\mathrm{U}(\mathtt{l}) : \mathrm{Ty}} \qquad \frac{\mathtt{l} : \mathrm{Lvl} \qquad \mathsf{a} :_{\mathsf{p}} \mathrm{Tm}(\mathrm{U}(\mathtt{l}))}{\mathrm{El}(\mathsf{a}) : \mathrm{Ty}} \qquad \frac{\mathtt{i} : \mathrm{Lvl} \qquad \mathtt{l} : \mathrm{Lvl} \qquad \mathtt{i} \equiv \mathrm{S}(\mathtt{l})}{\mathrm{u}(\mathtt{l}) : \mathrm{Tm}(\mathrm{U}(\mathtt{i}))} \qquad \mathrm{El}(\mathrm{u}(\mathtt{l})) \longmapsto \mathrm{U}(\mathtt{l})$$

$$\frac{\begin{array}{c}\mathtt{l} : \mathrm{Lvl} \qquad \mathsf{a} : \mathrm{Tm}(\mathrm{U}(\mathtt{l})) \\ x : \mathrm{Tm}(\mathrm{El}(\mathsf{a})) \vdash \mathsf{b} : \mathrm{Tm}(\mathrm{U}(\mathtt{l}))\end{array}}{\pi(\mathsf{a}, x.\mathsf{b}\{x\}) : \mathrm{Tm}(\mathrm{U}(\mathtt{l}))} \qquad\qquad \mathrm{El}(\pi(\mathsf{a}, x.\mathsf{b}\{x\})) \longmapsto \Pi(\mathrm{El}(\mathsf{a}), x.\mathrm{El}(\mathsf{b}\{x\}))$$

Cumulativity

For now, in the above theory we can only find a code decoding to $\Pi(\mathrm{El}(a), x.\mathrm{El}(b))$ if a and b live in the same universe. In order to fix this, we could define an heterogeneous version of π allowing for a and b to be in different universes, in which case the code for the type $\Pi(\mathrm{El}(a), x.\mathrm{El}(b))$ would then live in the maximum of the two — see the file `mltt-tarski-heterogeneous.bitts` in the implementation where this solution is developed. Here, we instead prefer to add *cumulativity* to the theory. This is done by adding a *lift* $\Uparrow$ mapping a code a from $\mathrm{U}(l)$ to $\mathrm{U}(\mathrm{S}(l))$, and a rewrite rule identifying its elements with the ones for a. Now we can form a code for $\Pi(\mathrm{El}(a), x.\mathrm{El}(b))$ by simply lifting the smaller code to the universe of the bigger one.

$$\frac{\mathtt{l} : \mathrm{Lvl} \qquad \mathsf{A} : \mathrm{Tm}(\mathrm{U}(\mathtt{l}))}{\Uparrow(\mathsf{a}) : \mathrm{Tm}(\mathrm{U}(\mathrm{S}(\mathtt{l})))} \qquad\qquad \mathrm{El}(\Uparrow(\mathsf{a})) \longmapsto \mathrm{El}(\mathsf{a})$$

Some authors consider a stronger version of cumulativity in which the lift operator $\Uparrow$ commutes with all codes [Ass14, Ste19, Kov22]. This can be obtained by adding the following rewrite rule. Note that this is the first rewrite rule we consider which is not headed by a destructor, illustrating the usefulness of not imposing rules to be of this shape.

$$\Uparrow(\pi(\mathsf{a}, x.\mathsf{b}\{x\})) \longmapsto \pi(\Uparrow(\mathsf{a}), x.\Uparrow(\mathsf{b}\{x\}))$$

When considering the above rule, the rewrite system is not orthogonal anymore, and a more complex proof of confluence is required. We separate the rewrite system into $\mathcal{R}_1 \cup \mathcal{R}_2$, where $\mathcal{R}_1$ contains the rules $\mathrm{El}(\pi(\mathsf{a}, x.\mathsf{b}\{x\})) \longmapsto \Pi(\mathrm{El}(\mathsf{a}), x.\mathrm{El}(\mathsf{b}\{x\}))$ and $\mathrm{El}(\Uparrow(\mathsf{a})) \longmapsto \mathrm{El}(\mathsf{a})$ and $\Uparrow(\pi(\mathsf{a}, x.\mathsf{b}\{x\})) \longmapsto \pi(\Uparrow(\mathsf{a}), x.\Uparrow(\mathsf{b}\{x\}))$, and $\mathcal{R}_2$ contains all the other rewrite rules. Then $\mathcal{R}_2$ is confluent by orthogonality, and $\mathcal{R}_1$ is confluent because it is strongly normalizing and all critical pairs close [MN98]. Because $\mathcal{R}_1$ and $\mathcal{R}_2$ are left-linear and there are no critical pairs between them, we conclude that their union is confluent by Van Oostrom & Van Raamsdonk's orthogonal combinations criterion [vOvR94].

Universe polymorphism

With our current theory we can write the polymorphic identity function at the base universe $\lambda(A.\lambda(x.x)) : \mathrm{Tm}(\Pi(\mathrm{U}(0), a.\Pi(\mathrm{El}(a), \_.\mathrm{El}(a))))$, but there is no way of writing an identity function that can be used with types in all universes. This can be achieved by extending our theory with *universe polymorphism* [Kov22, BCDE23, ST14], in which we add a type former for quantifying over levels. We can then write the universe-polymorphic identity function $\Lambda(i.\lambda(A.\lambda(x.x))) : \mathrm{Tm}(\forall(i.\Pi(\mathrm{U}(i), a.\Pi(\mathrm{El}(a), \_.\mathrm{El}(a)))))$.

$$\frac{i : \mathrm{Lvl} \vdash \mathsf{A} : \mathrm{Ty}}{\forall(i.\mathsf{A}\{i\}) : \mathrm{Ty}} \qquad\qquad \frac{i : \mathrm{Lvl} \vdash \mathsf{A} : \mathrm{Ty} \qquad i : \mathrm{Lvl} \vdash \mathsf{t} : \mathrm{Tm}(\mathsf{A}\{i\})}{\Lambda(i.\mathsf{t}\{i\}) : \mathrm{Tm}(\forall(i.\mathsf{A}\{i\}))}$$

$$\frac{i : \mathrm{Lvl} \vdash \mathsf{A} : \mathrm{Ty} \qquad \mathsf{t} :_{\mathrm{p}} \mathrm{Tm}(\forall(i.\mathsf{A}\{i\})) \qquad \mathsf{l} : \mathrm{Lvl}}{\mathrm{inst}(\mathsf{t}, \mathsf{l}) : \mathrm{Tm}(\mathsf{A}\{\mathsf{l}\})} \qquad\qquad \mathrm{inst}(\Lambda(i.\mathsf{t}\{i\}), \mathsf{l}) \longmapsto \mathsf{t}\{\mathsf{l}\}$$

Coquand-style universes

Up until now we have only seen examples showing the use of Tarski-style universes, however in our framework we can also define *Coquand-style universes* [Coq13, KHS19, Kov22]. In this approach, we start by redefining the sorts Ty and Tm so that they become themselves stratified — note therefore that this example is the first which is not an extension of $\mathbb{T}^{\flat}_{\lambda\Pi}$.

$$\frac{}{\mathrm{Lvl}\ \mathrm{sort}} \qquad \frac{}{0 : \mathrm{Lvl}} \qquad \frac{\mathsf{l} : \mathrm{Lvl}}{\mathrm{S}(\mathsf{l}) : \mathrm{Lvl}} \qquad \frac{\mathsf{l} : \mathrm{Lvl}}{\mathrm{Ty}(\mathsf{l})\ \mathrm{sort}} \qquad \frac{\mathsf{l} : \mathrm{Lvl} \qquad \mathsf{A} : \mathrm{Ty}(\mathsf{l})}{\mathrm{Tm}(\mathsf{l}, \mathsf{A})\ \mathrm{sort}}$$

We then postulate once again a type for the universe U and a decoding function El, but instead of adding codes for each type former manually, we add a code constructor c. Two rewrite rules then state that decoding $\mathrm{c}(A)$ yields precisely A, and that coding $\mathrm{El}(a)$ yields precisely a, thus establishing a definitional isomorphism between the sort $\mathrm{Ty}(l)$ and the sort $\mathrm{Tm}(\mathrm{S}(l), \mathrm{U})$, for all $l : \mathrm{Lvl}$.

$$\frac{\mathsf{l} : \mathrm{Lvl}}{\mathrm{U} : \mathrm{Ty}(\mathrm{S}(\mathsf{l}))} \qquad \frac{\mathsf{l} : \mathrm{Lvl} \qquad \mathsf{A} : \mathrm{Ty}(\mathsf{l})}{\mathrm{c}(\mathsf{A}) : \mathrm{Tm}(\mathrm{S}(\mathsf{l}), \mathrm{U})} \qquad \frac{\mathsf{l} : \mathrm{Lvl} \qquad \mathsf{a} :_{\mathrm{p}} \mathrm{Tm}(\mathrm{S}(\mathsf{l}), \mathrm{U})}{\mathrm{El}(\mathsf{a}) : \mathrm{Ty}(\mathsf{l})} \qquad \begin{array}{l} \mathrm{El}(\mathrm{c}(\mathsf{A})) \longmapsto \mathsf{A} \\ \mathrm{c}(\mathrm{El}(\mathsf{a})) \longmapsto \mathsf{a} \end{array}$$

We can extend the theory with other type formers, such as function types. Note then that, not only the types A and B can be omitted, but also the level l.

$$\frac{\mathtt{l} : \mathrm{Lvl} \qquad \mathtt{A} : \mathrm{Ty}(\mathtt{l}) \qquad x : \mathrm{Tm}(\mathtt{l}, \mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty}(\mathtt{l})}{\Pi(\mathtt{A}, x.\mathtt{B}\{x\}) : \mathrm{Ty}(\mathtt{l})} \qquad \frac{\mathtt{l} : \mathrm{Lvl} \qquad \mathtt{A} : \mathrm{Ty}(\mathtt{l}) \qquad x : \mathrm{Tm}(\mathtt{l}, \mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty}(\mathtt{l}) \qquad x : \mathrm{Tm}(\mathtt{l}, \mathtt{A}) \vdash \mathtt{t} : \mathrm{Tm}(\mathtt{l}, \mathtt{B}\{x\})}{\lambda(x.\mathtt{t}\{x\}) : \mathrm{Tm}(\mathtt{l}, \Pi(\mathtt{A}, x.\mathtt{B}\{x\}))}$$

$$\frac{\mathtt{l} : \mathrm{Lvl} \qquad \mathtt{A} : \mathrm{Ty}(\mathtt{l}) \qquad x : \mathrm{Tm}(\mathtt{l}, \mathtt{A}) \vdash \mathtt{B} : \mathrm{Ty}(\mathtt{l}) \qquad \mathtt{t} :_{\mathrm{p}} \mathrm{Tm}(\mathtt{l}, \Pi(\mathtt{A}, x.\mathtt{B}\{x\})) \qquad \mathtt{u} : \mathrm{Tm}(\mathtt{l}, \mathtt{A})}{@(\mathtt{t}, \mathtt{u}) : \mathrm{Tm}(\mathtt{l}, \mathtt{B}\{\mathtt{u}\})} \qquad @(\lambda(x.\mathtt{t}\{x\}), \mathtt{u}) \longmapsto \mathtt{t}\{\mathtt{u}\}$$

We refer to the file `mltt-coquand.bitts` of the implementation in which this approach is further developed.

Finally, note that the rewrite system of this theory is not orthogonal, because the rewrite rules $\mathrm{El}(\mathrm{c}(\mathtt{A})) \longmapsto \mathtt{A}$ and $\mathrm{c}(\mathrm{El}(\mathtt{a})) \longmapsto \mathtt{a}$ create two critical paris. Fortunately, these critical pairs are *trivial*, so the rewrite system is *weakly orthogonal* and hence confluent [vOvR94].

Russell-style universes

Coquand-style universes are characterized by a definitional isomorphism $\mathrm{Tm}(\mathrm{S}(l), \mathrm{U}) \simeq \mathrm{Ty}(l)$, establishing that the two sorts are morally the same. We can however also go further and identify the two with a rewrite rule, yielding *Russell-style universes*. This is done by simply replacing the previous declarations of El, c and the associated rules with the rule $\mathrm{Tm}(\mathrm{S}(\mathtt{l}), \mathrm{U}) \longmapsto \mathrm{Ty}(\mathtt{l})$. We refer to the file `mltt-russell.bitts` of the implementation for more details.

9.5 Exceptional type theory

We have seen that our framework supports the definition of many features commonly found in dependent type theories. However, we can also define theories with features that are more unusual. Let us now see how to define in our framework a variant of Pédrot and Tabareau's Exceptional Type Theory [PT18], that extends MLTT with exceptions. Starting from $\mathbb{T}^{\flat}_{\lambda\Pi}$, we add a new sort Ex for exceptions, a default exception err and a constructor raise that allows us to raise an exception at any type. Note that raise renders the theory inconsistent, however in their paper Pédrot and Tabareau show how to define a parametricity layer to isolate a consistent subset of the language, in which exceptions have to be caught locally. Nevertheless, if one wishes to use the theory for programming instead of proving theorems, it is reasonable to drop this extra layer and work in a language where all types are inhabited (like almost all commonly used programming languages).

$$\frac{}{\mathrm{Ex}\ \mathsf{sort}} \qquad \frac{}{\mathrm{err} : \mathrm{Ex}} \qquad \frac{\mathtt{A} : \mathrm{Ty} \qquad \mathtt{e} : \mathrm{Ex}}{\mathrm{raise}(\mathtt{e}) : \mathrm{Tm}(\mathtt{A})}$$

We then must add rules for ensuring that destructors properly propagate the exceptions, such as the following one.

$$@(\mathrm{raise}(\mathsf{e}), \mathsf{u}) \longmapsto \mathrm{raise}(\mathsf{e})$$

We can then extend the theory with generalized eliminators for the positive types in order to allow for exception-catching. For instance, supposing we also have extended the theory with booleans — and added the associated rule $\mathrm{rec}_{\mathbb{B}}(\mathrm{raise}(\mathsf{e}), x.\mathsf{P}\{x\}, \mathsf{pt}, \mathsf{pf}) \longmapsto \mathrm{raise}(\mathsf{e})$ for propagating exceptions — we can add a new eliminator $\mathrm{catch}_{\mathbb{B}}$ in which raise is treated like a constructor of $\mathbb{B}$, allowing us to handle the raised exception.

$$\frac{\begin{array}{c}\mathsf{t} :_{\mathsf{p}} \mathrm{Tm}(\mathbb{B}) \qquad x : \mathrm{Tm}(\mathbb{B}) \vdash \mathsf{P} : \mathrm{Ty} \\ \mathsf{pt} : \mathrm{Tm}(\mathsf{P}\{\mathrm{true}\}) \qquad \mathsf{pf} : \mathrm{Tm}(\mathsf{P}\{\mathrm{false}\}) \qquad x : \mathrm{Ex} \vdash \mathsf{pe} : \mathrm{Tm}(\mathsf{P}\{\mathrm{raise}(x)\})\end{array}}{\mathrm{catch}_{\mathbb{B}}(\mathsf{t}, x.\mathsf{P}\{x\}, \mathsf{pt}, \mathsf{pf}, x.\mathsf{pe}\{x\}) : \mathrm{Tm}(\mathsf{P}\{\mathsf{t}\})}$$

$$\begin{aligned}
&\mathrm{catch}_{\mathbb{B}}(\mathrm{true}, x.\mathsf{P}\{x\}, \mathsf{pt}, \mathsf{pf}, x.\mathsf{pe}\{x\}) \longmapsto \mathsf{pt} \\
&\mathrm{catch}_{\mathbb{B}}(\mathrm{false}, x.\mathsf{P}\{x\}, \mathsf{pt}, \mathsf{pf}, x.\mathsf{pe}\{x\}) \longmapsto \mathsf{pf} \\
&\mathrm{catch}_{\mathbb{B}}(\mathrm{raise}(\mathsf{e}), x.\mathsf{P}\{x\}, \mathsf{pt}, \mathsf{pf}, x.\mathsf{pe}\{x\}) \longmapsto \mathsf{pe}\{\mathsf{e}\}
\end{aligned}$$

This theory can then be extended with various types, and we refer to the file `exceptional.bitts` of the implementation for more details.

We can also interface the exceptional theory with a pure one by restricting the eliminators, in the spirit of the Multiverse Type Theory (MuTT) [MMS+21]. For this, we redefine the theory by parametrizing Ty and Tm by a *mode* $\mathfrak{M}$, which can either be $\mathbb{P}$ (for pure) or $\mathbb{E}$ (for exceptional). The raise constructor is then restricted to only allow for raising exceptions when in the exceptional mode, and the only way for eliminating from types in the exceptional mode to the pure world is by using catching eliminators, ensuring that exceptions cannot be propagated. We refer to the file `exceptional-multiverse.bitts` of the implementation, where this approach is sketched.

9.6 Observational type theory

As our last example, we show how to define a variant of Altenkirch and McBride's *observational type theory (OTT)* [AMS07, PT22] in our framework. Starting from $\mathbb{T}^{\flat}_{\lambda\Pi}$, we begin by extending the theory with an *heterogeneous* equality type.

$$\frac{\mathsf{A} : \mathrm{Ty} \qquad \mathsf{a} : \mathrm{Tm}(\mathsf{A}) \qquad \mathsf{B} : \mathrm{Ty} \qquad \mathsf{b} : \mathrm{Tm}(\mathsf{B})}{\mathrm{Eq}(\mathsf{A}, \mathsf{a}, \mathsf{B}, \mathsf{b}) : \mathrm{Ty}}$$

The defining characteristic of OTT is that its equality $\mathrm{Eq}(A, t, A', t')$ is defined inductively on the structure of the types A and A'. For instance, a proof of $\mathrm{Eq}(\Pi(A, x.B), f, \Pi(A', x'.B'\{x'\}), f')$ should correspond exactly to a function mapping proofs of equality between t and t' to proofs of equality between $@(f, t)$ and $@(f', t')$. In the original formulation of OTT [AMS07] this correspondence was made by adding a rewrite rule explaining how to reduce $\mathrm{Eq}(\Pi(A, x.B), f, \Pi(A', x'.B'\{x'\}), f')$. Here we instead adopt the approach taken by Atkey [Atk18] and more recently by Pujet and Tabareau [PT24], in which one instead postulates symbols $\mathrm{Eq}\Pi_i$ and $\mathrm{Eq}\Pi_e$ for constructing and eliminating equality proofs between f and f'. Note that our support for omitting arguments is vital here to ensure that the numerous arguments do not get all recorded in the syntax.

$$\frac{\begin{array}{ccc} \mathsf{A} : \mathrm{Ty} & x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} & \mathsf{f} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \\ \mathsf{A}' : \mathrm{Ty} & x' : \mathrm{Tm}(\mathsf{A}') \vdash \mathsf{B}' : \mathrm{Ty} & \mathsf{f}' : \mathrm{Tm}(\Pi(\mathsf{A}', x'.\mathsf{B}'\{x'\})) \\ \multicolumn{3}{c}{x : \mathrm{Tm}(\mathsf{A}), x' : \mathrm{Tm}(\mathsf{A}'), y : \mathrm{Tm}(\mathrm{Eq}(\mathsf{A}, x, \mathsf{A}', x')) \vdash \mathsf{p} : \mathrm{Tm}(\mathrm{Eq}(\mathsf{B}\{x\}, @(\mathsf{f}, x), \mathsf{B}'\{x'\}, @(\mathsf{f}', x')))} \end{array}}{\mathrm{Eq}\Pi_i(xx'y.\mathsf{p}\{x, x', y\}) : \mathrm{Tm}(\mathrm{Eq}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}), \mathsf{f}, \Pi(\mathsf{A}', x'.\mathsf{B}'\{x'\}), \mathsf{f}'))}$$

$$\frac{\begin{array}{ccc} \mathsf{A} : \mathrm{Ty} & x : \mathrm{Tm}(\mathsf{A}) \vdash \mathsf{B} : \mathrm{Ty} & \mathsf{f} : \mathrm{Tm}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\})) \\ \mathsf{A}' : \mathrm{Ty} & x' : \mathrm{Tm}(\mathsf{A}') \vdash \mathsf{B}' : \mathrm{Ty} & \mathsf{f}' : \mathrm{Tm}(\Pi(\mathsf{A}', x'.\mathsf{B}'\{x'\})) \\ \multicolumn{3}{c}{\mathsf{p} :_{\mathrm{p}} \mathrm{Tm}(\mathrm{Eq}(\Pi(\mathsf{A}, x.\mathsf{B}\{x\}), \mathsf{f}, \Pi(\mathsf{A}', x'.\mathsf{B}'\{x'\}), \mathsf{f}'))} \\ \mathsf{x} : \mathrm{Tm}(\mathsf{A}) & \mathsf{x}' : \mathrm{Tm}(\mathsf{A}') & \mathsf{e} : \mathrm{Tm}(\mathrm{Eq}(\mathsf{A}, \mathsf{x}, \mathsf{A}', \mathsf{x}')) \end{array}}{\mathrm{Eq}\Pi_e(\mathsf{p}, \mathsf{x}, \mathsf{x}', \mathsf{e}) : \mathrm{Tm}(\mathrm{Eq}(\mathsf{B}\{\mathsf{x}\}, @(\mathsf{f}, \mathsf{x}), \mathsf{B}'\{\mathsf{x}'\}, @(\mathsf{f}', \mathsf{x}')))}$$

If we add a Tarski-style universe U to the theory, we would then like to allow for transporting a term from $\mathrm{El}(a)$ to $\mathrm{El}(b)$ when we have a proof of $\mathrm{Eq}(\mathrm{U}, a, \mathrm{U}, b)$. In OTT, this is achieved by adding a cast operator to the theory. Note that the arguments a and b could be recovered from p, but they cannot be omitted because they are needed to know how to reduce casts.

$$\frac{\mathsf{a} :_{\mathrm{p}} \mathrm{Tm}(\mathrm{U}) \qquad \mathsf{b} : \mathrm{Tm}(\mathrm{U}) \qquad \mathsf{p} : \mathrm{Tm}(\mathrm{Eq}(\mathsf{a}, \mathrm{U}, \mathsf{b}, \mathrm{U})) \qquad \mathsf{t} : \mathrm{Tm}(\mathrm{El}(\mathsf{a}))}{\mathrm{cast}(\mathsf{a}, \mathsf{b}, \mathsf{p}, \mathsf{t}) : \mathrm{Tm}(\mathrm{El}(\mathsf{b}))}$$

$$\mathrm{cast}(\pi(\mathsf{a}, x.\mathsf{b}\{x\}), \pi(\mathsf{a}', x'.\mathsf{b}'\{x'\}), \mathsf{p}, \mathsf{t}) \longmapsto \begin{array}{l} \lambda(x'.\ \ \mathsf{let}\ e := \mathrm{EqU}^{\pi,\pi}_{e1}(\mathsf{p})\ \mathsf{in} \\ \qquad \mathsf{let}\ x := \mathrm{cast}(\mathsf{a}', \mathsf{a}, e, x')\ \mathsf{in} \\ \qquad \mathrm{cast}(\mathsf{b}\{x\}, \mathsf{b}\{x'\}, \mathrm{EqU}^{\pi,\pi}_{e2}(\mathsf{p}, x, x', e), @(\mathsf{t}, x))) \end{array}$$

We refer to the file `ott.bitts` of the implementation in which this construction is worked out in detail. We also provide a second variant of OTT in the file `ott-2.bitts`, this time using an homogeneous equality, in the style of Pujet and Tabareau [PT22].

Chapter 10

BiTTs: An Implementation of our Framework

The bidirectional type system of Chapter 8 has been implemented in the tool BiTTs (for Bidirectional Type Theories), available at

https://github.com/thiagofelicissimo/BiTTs

and which we present in this section. We start by illustrating how the tool can be used in practice, and then briefly discuss some aspects of the implementation.

10.1 A quick introduction to the tool

As we have seen, our framework is not designed to target a specific type theory, but instead a whole class of bidirectional theories. Therefore, in order to use our tool, the first step is to specify the bidirectional theory we want to work in. This is done with the commands `sort`, `constructor`, `destructor` and `equation` which specify respectively sort, constructor, destructor and rewrite rules. For instance, the bidirectional theory $\mathbb{T}^{\flat}_{\lambda\Pi}$ can be defined with the following declarations. Note that metavariable contexts are delimited by parentheses, principal arguments are delimited by square brackets, and that we support unicode characters in names.

```
sort Ty ()
sort Tm (A : Ty)
constructor Π () (A : Ty, B{x : Tm(A)} : Ty) : Ty
constructor λ (A : Ty, B{x : Tm(A)} : Ty) (t{x : Tm(A)} : Tm(B{x})) : Tm(Π(A, x. B{x}))
destructor @ (A : Ty, B{x : Tm(A)} : Ty) [t : Tm(Π(A, x. B{x}))] (u : Tm(A)) : Tm(B{u})
equation @(λ(x. t{x}), u) --> t{u}
```

By Theorem 8.3, the type-checker implemented is sound with respect to the type system of Figure 7.4, as soon as the specified bidirectional theory is valid. Checking this hypothesis is mostly left to the user, however the implementation helps to check some of the required conditions. First, it verifies automatically condition **(B)**, and it reports on any critical pairs, so that if there are none then condition **(II)** is satisfied by orthogonality. The implementation also tries to verifies condition **(I)**, however this check relies on the hypothesis that the theory satisfies confluence and subject reduction *incrementally*, meaning that each prefix of the theory should also verify these properties.[1] Thankfully, for most theories the rewrite system is orthogonal, ensuring that any prefix of it is also confluent, and the verification that rewrite rules preserve typing is in most cases modular (see the discussion at the end of Example 7.8).

Once the theory is specified, we can start writing and typechecking terms in the implementation. For instance, supposing we have also added a Tarski-style universe `U`, we can check the following definition of the polymorphic identity function.

```
let idU : Tm(Π(U, a. Π(El(a), _. El(a)))) := λ(a. λ(x. x))
```

To type-check this definition, the tool first verifies that the sort given in the annotation is well-typed, and then type-checks the body of the definition against the sort. If all the steps succeed, the identifier is added to a global scope of top-level definitions and becomes available to be used in the rest of the file.

The implementation also supports local let definitions, as illustrated in the following example, which also shows how sort ascriptions can be used.

```
let redex' : Tm(ℕ) :=
    let ty : Ty := Π(ℕ, _. ℕ) in
    @(λ(x. x) :: Tm(ty), 0)
```

Finally, we also provide commands for evaluating terms to normal form and asserting that two terms are definitionally equal. For instance, assuming we have defined factorial, we can use these commands to compute the factorial of 3 and check that it is equal to 6.

```
let fact3 : Tm(ℕ) := @(fact, S(S(S(0))))
evaluate fact3
let 6 : Tm(ℕ) := S(S(S(S(S(S(0))))))
assert fact3 = 6
```

The implementation also comes with many theories that can be defined in the framework, along with some examples of terms written in these theories. These can be found in the directory `examples/`, and most of them are discussed in Chapter 9.

[1] The reason is that we use an extension of the system of Figure 8.3 to check the typing judgments of Figure 7.5, however its soundness requires the bidirectional theory to be valid.

10.2 The implementation

The core of the implementation can be separated into two main parts: the type-checking and the normalization algorithms. The type-checking algorithm follows closely the bidirectional system described in Chapter 8 with only minor differences, so we will not discuss it here.

Regarding normalization, we have implemented it by employing an untyped variant of *Normalization by Evaluation (NbE)*, inspired by the works of Coquand [Coq96], Abel [Abe13] and Kovacs [Kov23]. In NbE, terms are evaluated into a separate syntax of runtime values, in which binders are represented by closures and free variables by unknowns. This evaluation roughly corresponds to a call-by-value reduction to a head normal form. Values can then be compared for equality by entering closures and recursively evaluating and comparing their bodies. One of the benefits of this approach is that, by using de Bruijn indices in the syntax of regular terms but de Bruijn *levels*[2] in the syntax of values, we completely avoid the need of implementing substitution or index-shifting functions, which are often complicated and inefficient — see for instance the discussion by Gratzer *et al.* [GSB19], or the lecture given by Sterling [Ste22a].

Preliminary tests suggest that our implementation of normalization has good performances: we can compute factorial of 8 with unary natural numbers in around half a second, whereas the same test leads Coq to crash with a stack overflow and Agda to run for more than a minute before we terminate the process. In the future, we plan to further test the general performances of our tool with larger and more realistic examples. In particular, we would like to compare it with typecheckers for Dedukti, and we expect that BiTTs' support for non-annotated syntaxes should allow for shorter typechecking times.

[2] Whereas indices count variables starting from the right of the context, levels count them starting from the left, so the variable x in $x : T, y : U \vdash x : T$ is represented as 1 when using indices but 0 when using levels.

Chapter 11

Perspectives on Generic Bidirectional Typing

In this Part II we have given a logical framework for specifying theories with their usual non-annotated presentations, and then showed how our notion of theory could be refined with bidirectionality, allowing us to give in particular a *generic* treatment of bidirectional typing and to explain why the omission of some arguments does not jeopardize decidability of typing. Our main results, Theorems 8.3 and 8.4, establish an equivalence between declarative and bidirectional type systems for the whole class of valid bidirectional theories, as defined in Sections 8.1 and 8.5. The decidability of the bidirectional type system (Theorem 8.5) allowed for its implementation in the tool BiTTs, which has been used in practice with multiple theories. Compared to other theory-independent typecheckers, such as Dedukti and Andromeda [HB23, BK22], BiTTs' support for non-annotated syntaxes can allow for better performances, making it a good candidate for cross-checking real proof libraries.

11.1 Related work

Our framework draws much inspiration from other proposals, such as GATs/QIITs [Car86, AK16b, KKA19], SOGATs [Uem21], FTTs [HB23], and logical frameworks such as Dedukti [BDG+23] and Harper's Equational LF [Har21a]. However, we differ from these works by supporting non-annotated syntaxes, and enforcing a constructor/destructor separation of schematic typing rules in our bidirectional theories (both of which seem to be important ingredients for bidirectional typing).

Another point of divergence from these frameworks is that most of them allow the use of arbitrary equations when defining the definitional equality of theories. However, it then becomes hard to give an implementation, as it would require deciding arbitrary equational theories. We instead follow the approach of Dedukti of supporting only rewrite rules, which allows us to decide the definitional equality of theories in a uniform manner, and made it possible to implement our framework. A different approach is taken in Andromeda, an implementation of FTTs, where one also allows for extensionality rules [BK22]. They however provide no proof of completeness for their equality-checking algorithm.

Our proposal also draws inspiration from the works of McBride, a main advocate of dependent bidirectional typing. His ongoing work on a framework for bidirectional typing [McB18, McB22] shares many similarities with ours, for instance by adopting a constructor/destructor separation of rules. However, an important difference with our framework is that he takes the bidirectional type system as the definition of the theory. Therefore, there is no discussion on how to show soundness and completeness with respect to a declarative system, as the bidirectional one is the only type system defined in his setting. This approach differs from most of the literature on dependent bidirectional typing [AC05, LB21, AA11, ACP11, AVW17], in which one first defines the type theory by a "platonic" declarative type system and then shows it equivalent to a bidirectional system which can be implemented. Finally, this choice also makes the metatheoretic study of theories quite different from what we have done here: for instance, even to be able to state subject reduction for the bidirectional system, the notion of reduction has to be updated to take ascriptions into account.

Another work from which ours drew inspiration is the one of Reed [Ree08], who proposes a variant of the Edinburgh Logical Framework in which arguments can be omitted. Crucially, these arguments are not elaborated through global unification, but instead locally recovered by annotating each declaration with modes to guide a bidirectional algorithm. However, his framework does not allow for extending the definitional equality, meaning that one cannot define dependent type theories with non-trivial equalities directly, but instead has to encode its derivations trees. This also means that his system does not need to deal with some complications that arise in our more general setting, such as matching modulo.

Finally, concurrently to our work, Chen and Ko [CK24] have proposed a framework for simply typed bidirectional typing. They also define declarative and bidirectional systems and establish a correspondence between them. Compared to our work, their restriction to simple types removes many of the complexities that appear with dependent type theories. For instance, while their types are first-order terms with no notion of computation or typing, our sorts are higher-order terms considered modulo a set of rewrite rules and subject to typing judgments, making the process of recovering missing arguments much more intricate. The restriction to simple types also rules out examples like the ones presented in Chapter 9, given that they are all dependent type theories. They however provide an impressive formalization of all their results in Agda.

11.2 Future work

The most important omission that we would like to address in future work is that of type-directed equality rules, which are needed for handling η-laws and definitional proof irrelevance. As mentioned in Section 11.1, our choice of supporting only rewrite rules was motivated by the fact that they allow for deciding the definitional equality in a uniform way, which made it possible to implement our framework. Indeed, as long as a rewrite system $\mathcal{R}$ is both confluent and strongly normalizing, computing and comparing normal forms is a complete equality-checking algorithm, regardless of any other specificity of $\mathcal{R}$. In contrast, even if it is well known how to design complete equality checking algorithms for specific theories with type-directed equalities [AOV18], doing so in a general setting like ours seems to be an important challenge. We could take inspiration from the customizable equality-checking algorithm implemented in Andromeda [BK22]. However, as previously mentioned, their algorithm is not proven complete, so further research in this direction seems to be required.

Moreover, in this work we chose to adopt the style of bidirectional typing most preminent in the literature [McB18, DK21, AA11, Coq96, Nor07, GSB19], in which constructors always check and destructors always infer. Yet, some proposals take an alternative approach by adding enough annotations to the syntax so that all terms can always be inferred [LB22, HHP93, Pol92]. For instance, in this approach abstractions are annotated with their domains, and their bidirectional rule becomes:

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \qquad \Gamma, x : A \vdash t \Rightarrow B}{\Gamma \vdash \lambda x : A.t \Rightarrow \Pi x : A.B}$$

Because this rule is inferring, we could try to interpret it in our framework as a destructor rule, however it violates two of our assumptions: (1) that the principal argument should always be the first non-erased argument, and (2) that the principal argument should be inferred in the empty context. In general, the bidirectional approach in which all terms infer seems to be less symmetric and thus less amenable to a generic treatment, like the one we gave here. We actually tried to also acommotated this style in a preliminary version of this work, but the definitions turned out to be too complicated and difficult to handle. Still, we do not rule out that our framework could be adapted in the future to support more liberal bidirectional rules in a cleaner way.

Finally, one could argue that our choice of declarative type system is not "declarative" enough, as some authors prefer more abstract definitions for specifying what type theories are. For instance, the point of view that syntax should correspond to the initial model (for some notion of semantics) often leads one to consider fully anotated terms with typed equality (or even quotiented terms [AK16b]), whereas our declarative type system uses non-annotated terms and untyped equality. In the future, we would like to investigate if our results could be adapted to such a setting, for instance by considering a variant of Uemura's SOGATs [Uem21] for the declarative type system. In this setting, our bidirectional system would be adapted into an elaboration algorithm, producing core fully annotated syntax from the user-friendly bidirectional one — similarly to [GSA$^+$22].

Part III

Sharing Proofs with Predicative Systems

Chapter 12

Introduction to Part III

In Parts I and II we have studied logical frameworks from a more theoretical perspective, showing how one can prove encodings to be conservative and then integrating bidirectional typing to allow for eliminating some annotations. But we also saw in this thesis' introduction that frameworks like Dedukti are actually developed with practical applications in mind. In particular, a long-term goal of the Dedukti project is to improve the *interoperability* of proof assistants. Let us discuss this in more detail.

Interoperability of proof assistants

As discussed before, an important achievement of the research community in logic is the invention of proof assistants. Such tools allow for interactively writing proofs, which are then checked automatically and can then be reused in other developments. Proof assistants do not only help mathematicians make sure that their proofs are indeed correct, but are also used to verify the correctness of safety-critical software.

Unfortunately, a proof written in a proof assistant cannot be directly reused in another one, which makes each tool isolated in its own library of proofs. This is specially the case when considering two proof assistants with incompatible logics, as in this case simply translating from one syntax to another would not work. Therefore, in order to share proofs between systems it is often required to do logical transformations.

A naïve approach to share proofs from a proof assistant A to a proof assistant B is to define a transformation acting directly on the syntax of A and then implement it using the codebase of A. However, this code would be highly dependent on the implementation of A and can easily become outdated if the codebase of A evolves. Moreover, if there is another proof assistant A' whose logic is very similar to the one of A then this transformation would have to be implemented once again in order to be used with A' — the translation is *implementation-dependent*.

Logical Frameworks & Dedukti

A better solution is to instead first define the logics of all proof assistants in a logical framework, so that proof transformations can be defined uniformly *inside* it. This way, such transformations do not depend on the implementations anymore, but instead on the *logics* that are implemented.

The logical framework Dedukti is a good candidate for a system where multiple logics can be encoded, allowing for logical transformations to be defined uniformly inside it. Indeed, first, the framework was already shown to be sufficiently expressive to define (considerable fragments of) the logics of many proof assistants [BDG+23]. Moreover, previous works have shown how proofs can be transformed inside Dedukti. For instance, Thiré [Thi18] describes a transformation to translate a proof of Fermat's Little Theorem from the Calculus of Inductive Constructions to Higher Order Logic (HOL), which can then be exported to multiple proof assistants such as HOL-Light, PVS, Lean, etc. Géran [Gé] also used Dedukti to export the formalization of Euclid's Elements Book 1 in Coq [BNW19] to several proof assistants.

(Im)Predicativity

One of the challenges in proof interoperability is sharing proofs coming from impredicative proof assistants (the majority of them) with predicative ones such as Agda. Indeed, impredicativity, which states that propositions can refer to entities of arbitrary sizes, is a logical principle absent from predicative systems. It is therefore clear that any proof that uses impredicativity in an essential way cannot be translated to a predicative system. Nevertheless, one can wonder if most proofs written in impredicative systems really use impredicativity and, if not, how one could devise a way for detecting and translating them to predicative systems.

A predicativization transformation

In this Part III, we tackle this problem by proposing a transformation that tries to do precisely this. Our translation works by forgetting all the universe information of the initial impredicative term, and then trying to elaborate it into a predicative universe-polymorphic term as general as possible. The need for universe polymorphism arises from the fact that mapping each universe to a unique one in the target theory is in most cases insufficient — this is explained in details in Chapter 13.

Universe level unification

During the translation, we need to solve level unification problems which are generated when elaborating the impredicative term into a universe polymorphic one. We therefore develop a (partial) unification algorithm for the equational theory of universe levels. This is done by first giving a novel and complete characterization of which single equations admit a most general unifier (m.g.u.), along with an explicit description of a m.g.u. when it exists. This characterization is then employed in an algorithm implementing a *constraint-postponement* strategy: at each step, we look for an equation admitting a m.g.u. and solve it while applying the obtained substitution to the other equations, in the hope of bringing new ones to the fragment admitting a m.g.u.

The given algorithm is *partial* in the sense that, when the unification problem is not a singleton, it may fail to find a m.g.u. even in cases where there is one — see for instance Example 17.1. We then propose a second unification algorithm, this time proven to be complete, but under the hypothesis that two conjectures from max-plus algebra are true. As further explained in Chapter 17, the algorithm works by reducing the problem of unification to the one of finding finite bases of solutions for two linear problems in a certain max-plus algebra, very similarly to Baader's algorithm for commutative theories [BS01] (the precise relationship is discussed in Chapter 19). Finally, we then show how the first constraint-postponement-based algorithm can be used in conjunction with this second algorithm, optimizing the computation of unifiers.

The implementation

Our predicativization algorithm was implemented on top of the DkCheck type-checker for Dedukti with the tool Predicativize (available at `https://github.com/Deducteam/predicativize`), allowing for the translation of proofs inside Dedukti. Our tool works in a semi-automatic manner: most of the translation is handled by the proposed algorithm, yet some intermediate steps that are harder to automate currently require some user intervention. These translated proofs can then be exported to Agda, the main proof assistant based on predicative type theory.

Translating Matita's arithmetic library

The tool has been used to translate to the proof assistant Agda the *whole* of Matita's arithmetic library, making many important mathematical developments available to Agda users. In particular, this work has led to (as far as we know) the first ever proofs in Agda of Fermat's Little Theorem, stating that for $p \in \mathbb{N}$ prime and $n \in \mathbb{N}$ coprime to p we have n^{p-1} equal to 1 modulo p, and of Bertrand's Postulate,[1] stating that for all positive $n \in \mathbb{N}$ one can always find a prime number p with $n < p \leq 2n$.

The proof of Bertrand's Postulate in Matita had even been the subject of a whole journal publication [AR12], evidencing its complexity and importance. Thanks to Predicativize, the same hard work did not have to be repeated to make it available in Agda, as the transformation allowed the translation of the whole proof without *any* need of specialist knowledge about it.

Related version

A preliminary version of this work was published in the proceedings of the 31st EACSL Annual Conference on Computer Science Logic [FBB23]. The version given here is closer

[1] Which, despite its name, is actually a theorem and not a postulate.

to (while not being exactly the same) an extended version to appear in the journal Logical Methods in Computer Science (LMCS), and contains a number of improvements, among which are the following:

1. A main novelty with respect to [FBB23] is that we provide a complete characterization of when a single equation between universe levels admit a m.g.u., along with an explicit description of such a m.g.u. This characterization then allows us to give a better algorithm for level unification, which in particular is complete for singleton problems, whereas the original algorithm is not. While this algorithm is still not complete in general, we then also give a second unification algorithm, this time with a proof of completeness, but which relies on two conjectures on max-plus algebra that are left for future work.[2]

2. The Church-Rosser modulo proof of $\mathbb{T}^{\forall}_{\mathbf{P}}$ in our preliminary work relied on the *ad hoc* restriction that level variables could only be replaced by universe levels, in order to avoid interactions between the equational theory of levels and the rewrite rules. We make this condition more precise by proposing a version of DEDUKTI tailored for external equational theories, in which we employ a form of *confinement*. This technique, first proposed by Assaf *et al.* [ADJL17], allows one to isolate a first-order subset of terms from the global higher-order syntax.

3. Finally, most of the text has been rewritten in order to improve the presentation.

[2] This second unification algorithm is a new contribution, that does not appear in the LMCS paper

Chapter 13

A First Informal Look at Proof Predicativization

In this informal chapter we present the problem of proof predicativization and discuss the challenges that arise through the use of examples. Even though the examples might be simplistic, they showcase real problems we found during our first predicativization attempt of Fermat's Little Theorem — some of them being already noted by Delort [Del20].

Preliminaries

We model the problem of predicativization in Dedukti as the task of translating from an impredicative theory $\mathbb{T}_{\mathbf{I}}$ to a predicative one $\mathbb{T}_{\mathbf{P}}$, specified by instantiating the Dedukti theory for PTSs in Figure 5.2 with the following specifications.[1]

$$\begin{aligned}
&\mathscr{L}_{\mathbf{I}} := \{\Omega, \square\} && \mathscr{L}_{\mathbf{P}} := \mathbb{N}\\
&\mathscr{A}_{\mathbf{I}} := \{(\Omega, \square)\} && \mathscr{A}_{\mathbf{P}} := \{(n, n+1) \mid n \in \mathbb{N}\}\\
&\mathscr{R}_{\mathbf{I}} := \{(\Omega,\Omega,\Omega), (\square,\Omega,\Omega), (\square,\square,\square)\} && \mathscr{R}_{\mathbf{P}} := \{(n, m, \mathsf{max}\{n,m\}) \mid n, m \in \mathbb{N}\}
\end{aligned}$$

Note that in the theory $\mathbb{T}_{\mathbf{I}}$ we have $(\square, \Omega, \Omega) \in \mathscr{R}_{\mathbf{I}}$, and thus for $\Gamma \vdash A : \mathrm{Tm}_{\top}\ \mathrm{U}_{\square}$ and $\Gamma, x : \mathrm{Tm}_{\square}\ A \vdash B : \mathrm{Tm}_{\square}\ \mathrm{U}_{\Omega}$ we have $\Gamma \vdash \Pi_{\square,\Omega}\ (x.A)\ B : \mathrm{Tm}_{\square}\ \mathrm{U}_{\Omega}$. Therefore, the universe U_{Ω} is closed under dependent products indexed over types in $\mathrm{U}_{\square}$, a larger universe, so $\mathbb{T}_{\mathbf{I}}$ is indeed an impredicative theory. Finally, we note that $\mathbb{T}_{\mathbf{P}}$ is a subtheory of the one implemented in Agda, whereas $\mathbb{T}_{\mathbf{I}}$ is a subtheory of the ones implemented in Coq[2], Matita, Isabelle, etc, justifying why they are of interest.

In order for the examples that will follow to be readable, it is indispensable that we adopt a more informal and lighter notation for terms in $\mathbb{T}_{\mathbf{I}}$ and $\mathbb{T}_{\mathbf{P}}$: we will write

[1] Recall that the theory of Figure 5.2 is actually only defined for finite specifications. However, as explained in Remark 5.3, we can abuse the definitions and still consider the theory $\mathbb{T}_{\mathbf{P}}$, specially given that we are in an informal chapter.

[2] There the impredicative universe U_{Ω} is referred to as Prop, as its elements are most often seen as propositions.

$\Pi_{l,l'}\ A\ (x.B)$ as $\Pi x : A.B$ (or as $A \rightsquigarrow B$ when $x \notin \mathrm{fv}(B)$), and $@_{l,l'}\ A\ (x.B)\ t\ u$ as $t_{@}u$, and $\lambda_{l,l'}\ A\ (x.B)\ t\ u$ as $\lambda x.t$, and $\mathrm{Tm}_l\ A$ as $\mathrm{Tm}\ A$. Note that, unlike in Part II, this is just an informal notation used in examples, and in the actual DEDUKTI syntax all the omitted arguments have to be written.[3]

In order to properly model the problem of predicativization, we also have to introduce a notation for *local signatures* Φ, made of entries of the form $f : T$ and $f : T := t$. We then write $\mathbb{T} \ltimes \Phi$ for the *extension* of $\mathbb{T}$ with Φ, defined by

$$\begin{aligned} \mathbb{T} \ltimes (\cdot) &:= \mathbb{T} \\ \mathbb{T} \ltimes (\Phi,\ f : T) &:= \mathbb{T} \ltimes \Phi,\ f : T \\ \mathbb{T} \ltimes (\Phi,\ f : T := t) &:= \mathbb{T} \ltimes \Phi,\ f : T,\ f \longmapsto t \end{aligned}$$

and we say that Φ is *well-typed* in $\mathbb{T}$ when we can derive $\mathbb{T} \vdash \Phi$ using the following rules:

$$\frac{}{\mathbb{T} \vdash \cdot} \qquad \frac{\mathbb{T} \vdash \Phi \qquad \mathbb{T} \ltimes \Phi \triangleright \cdot \vdash T : s}{\mathbb{T} \vdash \Phi, f : T} \qquad \frac{\mathbb{T} \vdash \Phi \qquad \mathbb{T} \ltimes \Phi \triangleright \cdot \vdash t : T}{\mathbb{T} \vdash \Phi, f : T := t}$$

In the following, we write names of symbols in the local signature in sans serif black in order to distinguish them from symbols of the theory, whose names are still written in blue serif, following the conventions of Chapter 2.

Then, the problem of proof predicativization consists in defining a transformation such that, given a local signature Φ well-formed in $\mathbb{T}_{\mathbf{I}}$, we obtain a local signature Φ' well-formed in $\mathbb{T}_{\mathbf{P}}$ — a suitable transformation should of course preserve the structure of the statements in Φ, however we leave the precise relationship between Φ and Φ' vague at this point. Stated more informally, we would like to translate symbol declarations $f : T$ (which represent axioms) and symbol definitions $f : T := t$ (which also represent proofs) from $\mathbb{T}_{\mathbf{I}}$ to $\mathbb{T}_{\mathbf{P}}$. Note in particular that such a transformation is not applied to a single term but to a sequence of declarations and definitions, which can be related by dependency. This dependency, as we shall see, turns out to be a major issue for the translation.

1st try: A naïve translation

Now that our basic notions are explained, let us try to predicativize proofs. For our first step, consider a very simple development showing that for every object type A in U_Ω we can build a term in $A \rightsquigarrow A$ — this is actually just the polymorphic identity function for the universe U_Ω. Here we adopt an AGDA-like syntax to display entries of the local signature.

$$\begin{aligned} &\mathsf{id} : \mathrm{Tm}\ (\Pi A : \mathrm{U}_\Omega.A \rightsquigarrow A) \\ &\mathsf{id} := \lambda A.\lambda x.x \end{aligned}$$

[3] One then may wonder why not using the framework of Part II instead of DEDUKTI. The reason is very mundane: the work presented here had been done before the one of Part II.

To translate this simple development, the first idea that comes to mind is to define a mapping on universe levels: the level Ω is mapped to 0 and the level $\square$ is mapped to 1. If this mapping defined a *specification morphism*[4] then this transformation would always produce a well-typed definition in $\mathbb{T}_{\mathbf{P}}$ [Geu93, Lemma 4.2.6]. Unfortunately, it is easy to check that it does not define a specification morphism (worse, no function $\mathscr{L}_{\mathbf{I}} \to \mathscr{L}_{\mathbf{P}}$ defines a specification morphism). Nevertheless, this does not mean that it cannot produce something well-typed in $\mathbb{T}_{\mathbf{P}}$ in *some* cases. For instance, by applying it to id we get the following entry,[5] which is actually well-typed in $\mathbb{T}_{\mathbf{P}}$.

$$\begin{aligned}&\mathsf{id} : \mathrm{Tm}\ (\Pi A : \mathrm{U}_0 . A \rightsquigarrow A)\\ &\mathsf{id} := \lambda A.\lambda x.x\end{aligned}$$

This naïve approach however quickly fails when considering other cases. For instance, suppose now that one adds the following definition.

$$\begin{aligned}&\mathsf{id\text{-}to\text{-}id} : \mathrm{Tm}\ ((\Pi A : \mathrm{U}_\Omega . A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathrm{U}_\Omega . A \rightsquigarrow A))\\ &\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathrm{U}_\Omega . A \rightsquigarrow A)\end{aligned}$$

If we try to perform the same syntactic translation as before, we get the following result.

$$\begin{aligned}&\mathsf{id\text{-}to\text{-}id} : \mathrm{Tm}\ ((\Pi A : \mathrm{U}_0 . A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathrm{U}_0 . A \rightsquigarrow A))\\ &\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathrm{U}_0 . A \rightsquigarrow A)\end{aligned}$$

However, one can verify that this term is not well typed. Indeed, in the original term one quantifies over all types in U_Ω in the term $\Pi A : \mathrm{U}_\Omega . A \rightsquigarrow A$, and because of impredicativity this term stays at U_Ω. However, in $\mathbb{T}_{\mathbf{P}}$ quantifying over all elements of the universe U_0 in $\Pi A : \mathrm{U}_0 . A \rightsquigarrow A$ lifts the overall type of the term to U_1. As id expects a term of type U_0, the term $\mathsf{id}_{@}(\Pi A : \mathrm{U}_0 . A \rightsquigarrow A)$ is not well-typed.

2nd try: Elaborating proofs with Universo

The takeaway lesson from the first try is that impredicativity introduces a kind of *typical ambiguity*, as it allows us to put in a single universe U_Ω types which, in a predicative setting, would have to be stratified and placed in larger universes. Therefore, we should not translate every occurrence of Ω to 0 naively as we did, but try to compute for each occurrence of Ω some natural number n such that replacing it by n would produce a well-typed term in $\mathbb{T}_{\mathbf{P}}$. In other words, we should erase all universe level information and then *elaborate* it into a well-typed term in $\mathbb{T}_{\mathbf{P}}$.

[4] That is, a mapping of levels $\phi : \mathscr{L}_{\mathbf{I}} \to \mathscr{L}_{\mathbf{P}}$ such that $(l, l') \in \mathscr{A}_{\mathbf{I}}$ implies $(\phi(l), \phi(l')) \in \mathscr{A}_{\mathbf{P}}$ and $(l, l', l'') \in \mathscr{R}_{\mathbf{I}}$ implies $(\phi(l), \phi(l'), \phi(l'')) \in \mathscr{R}_{\mathbf{P}}$.

[5] Modulo the recomputation of some omitted levels.

Thankfully, performing such kind of transformations is exactly the goal of the tool Universo [Thi20]. To understand how it works, let us come back to the previous example. Universo starts here by replacing each level by a fresh schematic level representing a natural number.

$$\begin{aligned}
&\mathsf{id} : \mathrm{Tm}\ (\Pi A : \mathrm{U}_{i_1}.A \rightsquigarrow A)\\
&\mathsf{id} := \lambda A.\lambda x.x\\
&\mathsf{id\text{-}to\text{-}id} : \mathrm{Tm}\ ((\Pi A : \mathrm{U}_{i_2}.A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathrm{U}_{i_3}.A \rightsquigarrow A))\\
&\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathrm{U}_{i_4}.A \rightsquigarrow A)
\end{aligned}$$

Then, in the following step Universo tries to elaborate these into well-typed terms in $\mathbb{T}_{\mathbf{P}}$. To do so, it first tries to typecheck them and generates constraints in the process. These constraints are then given to a SMT solver, which is used to compute for each schematic level i a natural number so that the local signature is well-typed in $\mathbb{T}_{\mathbf{P}}$. For instance, applying Universo to our previous example produces the following local signature, which is indeed well-typed in $\mathbb{T}_{\mathbf{P}}$.

$$\begin{aligned}
&\mathsf{id} : \mathrm{Tm}\ (\Pi A : \mathrm{U}_{1}.A \rightsquigarrow A)\\
&\mathsf{id} := \lambda A.\lambda x.x\\
&\mathsf{id\text{-}to\text{-}id} : \mathrm{Tm}\ ((\Pi A : \mathrm{U}_{0}.A \rightsquigarrow A) \rightsquigarrow (\Pi A : \mathrm{U}_{0}.A \rightsquigarrow A))\\
&\mathsf{id\text{-}to\text{-}id} := \mathsf{id}_{@}(\Pi A : \mathrm{U}_{0}.A \rightsquigarrow A)
\end{aligned}$$

By using Universo it is possible to go much further than with the naïve method shown before. Still, this approach also fails when being employed with real libraries. To see the reason, consider the following minimum example, in which one uses id twice to build another element of the same type.

$$\begin{aligned}
&\mathsf{id}' : \mathrm{Tm}\ (\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A)\\
&\mathsf{id}' := \mathsf{id}_{@}(\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A)_{@}\mathsf{id}
\end{aligned}$$

If we repeat the same procedure as before, we get the following entries, which when type-checked generate unsolvable constraints.

$$\begin{aligned}
&\mathsf{id} : \mathrm{Tm}\ (\Pi A : \mathrm{U}_{i_1}.A \rightsquigarrow A)\\
&\mathsf{id} := \lambda A.\lambda x.x\\
&\mathsf{id}' : \mathrm{Tm}\ (\Pi A : \mathrm{U}_{i_2}.A \rightsquigarrow A)\\
&\mathsf{id}' := \mathsf{id}_{@}(\Pi A : \mathrm{U}_{i_3}.A \rightsquigarrow A)_{@}\mathsf{id}
\end{aligned}$$

The reason is that the application $\mathsf{id}_{@}(\Pi A : \mathrm{U}_{i_3}.A \rightsquigarrow A)_{@}\mathsf{id}$ forces i_1 to be both i_3 and $i_3 + 1$, which is of course impossible. Therefore, the takeaway lesson from this second try is that impredicativity does not only hide the fact that types need to be stratified, but also that they need to be usable at multiple levels of this stratification. Indeed, in our example we would like to use id both at type $\Pi A : \mathrm{U}_{i_3}.A \rightsquigarrow A$ and at type $\Pi A : \mathrm{U}_{i_3+1}.A \rightsquigarrow A$. In practice, when trying to translate libraries using Universo we found that at very early stages a translated proof or object was already needed at multiple universes at the same time, causing the translation to fail.

Our proposal: Universe-polymorphic elaboration

In order to properly compensate for the lack of impredicativity, we propose not to translate entries by fixing once and for all their universes, but instead to let them vary by using *universe polymorphism* [HP91, ST14]. This feature, present in some type theories (and also in the one of Agda [Tea]), allows defining terms containing universe variables, which can later be instantiated at various concrete universes.

Our translation will then work by first computing for each definition or declaration its set of constraints. However, instead of assigning concrete values to schematic levels, we perform *unification* which allows us to solve constraints in a symbolic way. The result will then be a universe polymorphic term, which will be usable at multiple universes when translating the next entries. In order to define this formally, in Chapter 15 we refine the target of our translation to a theory featuring Agda-style universe polymorphism, but for this we will first need to consider an extension of Dedukti with external equational theories, presented in the next chapter.

Chapter 14

Extending Dedukti with Confined Equational Theories

When defining the conversion of a dependent type theory, most equations can be split either as reduction rules or extensionality rules — though we do not consider theories with extensionality rules in this thesis. However, many recent type theories are also defined on top of some ad-hoc *external* equational theories containing equalities which can be much more arbitrary. This is the case for instance of type theory with Agda-style universe polymorphism, which is defined on top of an equational theory we dub the *theory of predicative universe levels*.

Because such theories often contain equations that cannot be oriented (such as commutativity), there is no hope of being able to express them with rewriting only, meaning they cannot be directly defined in vanilla Dedukti. One first solution would be to consider an extension of Dedukti with arbitrary undirected equations, however interactions between the external equational theory and the rewrite rules would not be ruled out *a priori*, and so proving confluence/Church-Rosser and deciding the conversion would become much more intricate. Whereas some criteria for Church-Rosser can still be found when the rewrite system is normalizing [Hue80, JK84, MN98] or when all equations are linear [Bla03, Fel24b], both assumptions prove to be often too strong four our use cases. For instance, as we will see, the theory of predicative levels uses equations which are non-linear, and because we prove Church-Rosser over pre-terms, rewriting is non-terminating because of the β-rule.

Thankfully, a solution to the aforementioned problem was found by Assaf *et al.* by proposing a variant of Dedukti with *confinement* [ADJL17]. This technique consists of isolating a first-order subset of the syntax, over which equations operate, from the global syntax of the framework, thereby limiting the interaction between rewrite rules and the equational theory. With this separation in place, the authors successfully managed to prove a Church-Rosser criterion that addressed the aforementioned problems.

In this chapter, we revisit Assaf *et al.*'s key idea by proposing yet another extension of Dedukti with confined equational theories along with a criterion for showing Church-Rosser in the theories defined therein. The need for improving Assaf *et al.*'s framework

$$\begin{array}{lrcll}
\boxed{\underline{\mathsf{Tm}}} \ni & \underline{t}, \underline{u} & ::= & \underline{x} \mid \underline{f}(\underline{t}_1, \dots, \underline{t}_k) & \text{where arity}(\underline{f}) = k \\
\boxed{\mathsf{Tm}} \ni & t, u, v, T, U & ::= & \cdots \mid \forall \underline{x}.T \mid \underline{x}.t \mid t\ \underline{u} & \\
\boxed{\mathsf{Ctx}} \ni & \Gamma, \Delta & ::= & \dots & \\
\boxed{\mathsf{Thy}} \ni & \mathbb{T} & ::= & \cdots \mid \mathbb{T}, \underline{f} \mid \mathbb{T}, \underline{t} \approx \underline{u} &
\end{array}$$

Figure 14.1: Raw syntax of the framework, extending the grammars of Figure 2.1

steams from many reasons, the main one being its lack of support for abstracting confined variables. Without such a construction, one is unable to encode universe polymorphism when using the confined layer for representing universe levels, as level variables can then never be abstracted.

While our extension was designed with the example of Agda-style universe polymorphism in mind, we suspect that many other interesting examples of type theories with external equational theories could be covered by this framework. Moreover, we think that extensions of Dedukti in this direction could allow in the future to cover the case of *cubical type theories* [CCHM18], which are defined over an external equational theory of De Morgan algebras.

14.1 The framework

Given a new set of *confined variables* $\underline{x}, \underline{y}, \dots \in \underline{\mathcal{V}}$ and a set of *confined function symbols* $\underline{f} \in \underline{\mathcal{F}}$ equipped with arities, the raw syntax of our extension of Dedukti is defined in Figure 14.1.[1] Elements of $\underline{\mathsf{Tm}}$ are called *confined terms*, which are first-order terms built from $\underline{\mathcal{F}}$ and $\underline{\mathcal{V}}$. The grammar of regular terms is then extended with quantification over confined variables $\forall \underline{x}.T$, abstraction of a confined variable $\underline{x}.t$ and application to a confined term $t\ \underline{u}$ — similarly to how first-order terms are added to proof terms and propositions in first-order logic [SU06, Section 8.7].

The notion of substitution is also extended to allow for entries of the form $\underline{t}/\underline{x}$, and the application of a substitution to a term t or confined term $\underline{t}$ is defined in the most straightforward way possible.

Because we now also have redexes of the form $(\underline{x}.t)\underline{u}$, we also consider the $\underline{\beta}$ rule, defined by $(\underline{x}.t)\underline{u} \longrightarrow_{\underline{\beta}} t[\underline{u}/\underline{x}]$. The global rewriting relation is thus updated to $\longrightarrow_{\beta\underline{\beta}\mathcal{R}}$, and we still write it as just $\longrightarrow$ when $\mathcal{R}$ is clear from the context.

In this setting, an *equational theory* is a set $\mathcal{E}$ of equations of the form $\underline{t} \approx \underline{u}$, whose closure under context and substitution is written $\simeq_{\mathcal{E}}$, or just $\simeq$ when $\mathcal{E}$ is clear from the

[1] There, the "…" mean that we recover the clauses from Figure 2.1.

$\boxed{\Gamma \vdash t : T}$

$$\textsc{ForAll}\quad \underline{x} \notin \mathsf{fv}(\Gamma)\,\frac{\Gamma \vdash T : s}{\Gamma \vdash \forall \underline{x}.T : s} \qquad \textsc{AbsC}\quad \underline{x} \notin \mathsf{fv}(\Gamma)\,\frac{\Gamma \vdash t : T}{\Gamma \vdash \underline{x}.t : \forall \underline{x}.T} \qquad \textsc{AppC}\quad \underline{\mathcal{F}}(\underline{u}) \subseteq \underline{\mathcal{F}}_{\mathbb{T}}\,\frac{\Gamma \vdash t : \forall \underline{x}.T}{\Gamma \vdash t\ \underline{u} : T[\underline{u}/\underline{x}]}$$

Figure 14.2: Typing rules of the framework, extending those of Figure 2.2

context. The notion of theory $\mathbb{T}$ is extended with equations $\underline{t} \approx \underline{u}$ and confined symbols $\underline{f}$ (which, as explained before, are assumed to be equipped with an arity), and we write $\mathcal{E}_{\mathbb{T}}$ for the equational theory defined by $\mathbb{T}$, and $\underline{\mathcal{F}}_{\mathbb{T}} \subseteq \underline{\mathcal{F}}$ for the confined symbols specified in $\mathbb{T}$. Given a theory $\mathbb{T}$, the typing judgment is then defined by the rules in Figure 14.2. Note that in rule Conv the conversion $\equiv$ defined by the theory now does not only contain $\longrightarrow_{\beta}$ and $\longrightarrow_{\mathcal{R}_{\mathbb{T}}}$, but also $\simeq_{\mathcal{E}_{\mathbb{T}}}$ and $\longrightarrow_{\underline{\beta}}$. Also remark that confined terms are subject to no typing constraints: in rule AppC we only require the set of confined symbols in $\underline{u}$, written $\underline{\mathcal{F}}(\underline{u})$, to be a subset of $\underline{\mathcal{F}}_{\mathbb{T}}$, meaning that only symbols declared in the theory can be used in confined terms.

The definition of well-typed theory is then straightforwardly extended to our new notion of theory, by simply ignoring equations and confined symbol declarations, similarly to how rewrite rules are handled in Figure 2.3. Finally, we also consider the notion of *local signature* Φ and the definitions of $\mathbb{T} \ltimes \Phi$ and $\mathbb{T} \vdash \Phi$, as specified in Chapter 13.

14.2 Basic metatheory

The basic properties of Proposition 2.1 are updated here in the following manner:

Proposition 14.1 (Basic metaproperties). *Let us write $\Gamma \sqsubseteq \Gamma'$ when Γ is a subsequence ofΓ'.*

Weakening *Suppose $\Gamma \sqsubseteq \Gamma'$ and $\mathbb{T} \triangleright \Gamma' \vdash$. Then $\mathbb{T} \triangleright \Gamma \vdash t : T$ implies $\mathbb{T} \triangleright \Gamma' \vdash t : T$.*

Substitution property *If $\mathbb{T} \triangleright \Gamma, x : U, \Gamma' \vdash t : T$ and $\mathbb{T} \triangleright \Gamma \vdash u : U$ then $\mathbb{T} \triangleright \Gamma, \Gamma'[u/x] \vdash t[u/x] : T[u/x]$. If $\mathbb{T} \triangleright \Gamma \vdash t : T$ and $\mathcal{F}(\underline{u}) \subseteq \mathcal{F}_{\mathbb{T}}$ then $\mathbb{T} \triangleright \Gamma[\underline{u}/\underline{x}] \vdash t[\underline{u}/\underline{x}] : T[\underline{u}/\underline{x}]$.*

Conversion in context *If $\mathbb{T} \triangleright \Gamma, x : U, \Gamma' \vdash t : T$ and $\mathbb{T} \triangleright \Gamma \vdash U' : s$ and $U \equiv U'$ then $\mathbb{T} \triangleright \Gamma, x : U', \Gamma' \vdash t : T$.*

In the following points, suppose that $\mathbb{T}$ is well-typed.

Validity *If $\mathbb{T} \triangleright \Gamma \vdash t : T$ then either $T = \mathsf{Kind}$ or $\mathbb{T} \triangleright \Gamma \vdash T : s$ for $s = \mathsf{Type}$ or Kind.*

We say that a rule $l \longmapsto r$ preserves typing in $\mathbb{T}$ whenever $\mathbb{T} \triangleright \Gamma \vdash l[\theta] : T$ implies $\mathbb{T} \triangleright \Gamma \vdash r[\theta] : T$, for every θ, Γ, T.

Subject reduction for $\mathcal{R}_{\mathbb{T}}$ *If every rule in $\mathcal{R}_{\mathbb{T}}$ preserves typing in $\mathbb{T}$, then $\mathbb{T} \triangleright \Gamma \vdash t : T$ and $t \longrightarrow_{\mathcal{R}_{\mathbb{T}}} t'$ implies $\mathbb{T} \triangleright \Gamma \vdash t' : T$.*

In the following points, suppose that injectivity of $\rightarrow$ and $\forall$ hold in $\mathbb{T}$, meaning that $(x : T) \rightarrow U \equiv (x : T') \rightarrow U'$ implies $T \equiv T'$ and $U \equiv U'$, and that $\forall \underline{x}.T \equiv \forall \underline{x}.T'$ implies $T \equiv T'$.

Subject reduction for β and $\underline{\beta}$ *If $\mathbb{T} \triangleright \Gamma \vdash t : T$ and $t \longrightarrow_{\beta\underline{\beta}} t'$ then $\mathbb{T} \triangleright \Gamma \vdash t' : T$.*

Proof. Similar to Proposition 2.1. ■

14.3 A simple criterion for Church-Rosser modulo

In Dedukti, we often require theories to be Church-Rosser — or equivalently, for them to be confluent. Because we now consider rewriting with undirected equational theories, in this setting the Church-Rosser property must be adapted into *Church-Rosser modulo*, stating that for $t \equiv u$ we have $t \longrightarrow^* t' \simeq u' \ {}^*\!\!\longleftarrow u$ for some t', u'.[2]

In order to show this important property in our framework, we propose the following simple criterion. We emphasize that the use of confinement is essential in its proof — see Remark 14.1 for a detailed discussion on this.

Theorem 14.1. *If $\mathcal{R}_{\mathbb{T}}$ is left-linear, confluent and its left-hand sides do not mention symbols from $\underline{\mathcal{F}}$, then $\mathcal{R}_{\mathbb{T}}$ with $\mathcal{E}_{\mathbb{T}}$ is Church-Rosser modulo.*

The above result is important for two reasons. First, like in vanilla Dedukti, the Church-Rosser property allows us to establish the injectivity of →, and here also of ∀, which then ensure that β and $\underline{\beta}$ satisfy subject reduction. Church-Rosser is also needed for showing that most rewrite rules preserve typing.

Second, from an implementation point of view, Theorem 14.1 ensures us that for deciding $t \equiv u$ we do not need *matching modulo* $\mathcal{E}_{\mathbb{T}}$, which means deciding if there is θ with $\underline{t} \simeq \underline{u}[\theta]$ for some given $\underline{t}, \underline{u}$, but instead we only need for the word problem of $\mathcal{E}_{\mathbb{T}}$ to be decidable and for $\longrightarrow$ to be strongly normalizing for the considered subset of terms. This is a point of departure with respect to Assaf *et al.*'s original criterion [ADJL17], that instead proves a weaker version of Church-Rosser modulo in which rewriting employs matching modulo — necessary in their case because, unlike here, rewrite rule left-hand sides are allowed to mention confined symbols.[3]

[2] One can also generalize confluence to *confluence modulo*, however this property is *not* equivalent to Church-Rosser modulo, which is actually stronger [BKdVT03, Remark 14.3.6]. Therefore, we will not be interested in proving confluence modulo in this thesis.

[3] It is also worth noting that the hypotheses of our criteria are, in general, incomparable. For instance, if Assaf *et al.*'s result allows for confined symbols in rules, it also forbids rewrite rules whose right-hand sides are abstractions or variables, a restriction that is not necessary in our case. Additionally, we consider our proof to be more elementary, whereas Assaf *et al.*'s proof requires complex tools like decreasing diagrams and nested critical pairs.

We consider the avoidance of matching modulo to be important because the equational theories $\mathcal{E}_{\mathbb{T}}$ are user-defined, so it would not be reasonable to ask users to also provide an $\mathcal{E}_{\mathbb{T}}$-matching algorithm for each specified theory. Indeed, designing such algorithms is generally highly non-trivial and requires specialist knowledge about $\mathcal{E}_{\mathbb{T}}$. On the other hand, asking for the word problem to be decidable is a much more reasonable assumption, after all if $\simeq$ is not decidable then there is no hope of deciding $\equiv$.

Finally, we note that the approach of avoiding matching modulo was initially promoted by Huet in his seminal work on rewriting [Hue80], though his criterion for achieving this employs different hypotheses from the ones we have here. In particular, he requires $\longrightarrow \circ \simeq$ to be terminating, which does not hold in our setting, given that we prove confluence/Church-Rosser at the level of untyped terms.

We now show Theorem 14.1 in two steps. We first start with the following criterion for showing Church-Rosser modulo in the setting of abstract rewriting. Recall that an *abstract equational rewrite system* $(\triangleright, \sim)$ is given by a binary relation $\triangleright \subseteq X^2$ and an equivalence relation $\sim \subseteq X^2$. Then $(\triangleright, \sim)$ is said to be *Church-Rosser modulo* if $x\ (\triangleright \cup \triangleleft \cup \sim)^*\ y$ implies $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft\ y$, where we write $\circ$ for composition of relations, and $\triangleleft$ for the inverse of $\triangleright$. Finally, we say that $\sim$ is a *simulation* for $\triangleright$ if $\sim \circ \triangleright$ is included in $\triangleright \circ \sim$.

Proposition 14.2. *Let $(\triangleright, \sim)$ be an abstract equational rewrite system. If $\triangleright$ is confluent and $\sim$ is a simulation for $\triangleright$, then $(\triangleright, \sim)$ is Church-Rosser modulo.*

Proof. If $x\ (\triangleright \cup \triangleleft \cup \sim)^*\ y$, then we have $x\ (\triangleright \cup \triangleleft \cup \sim)^n\ y$ for some n. We prove the result by induction on n, the base case being trivial. For the inductive step, we have

$$x\ (\triangleright \cup \triangleleft \cup \sim)^n\ z\ (\triangleright \cup \triangleleft \cup \sim)\ y$$

for some z. First note that by i.h. we have $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft\ z$. We now have three possibilities:

1. $z \triangleright y$: We have $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft \circ \triangleright\ y$, so by confluence we have $x \triangleright^* \circ \sim \circ \triangleright^* \circ\ {}^*\!\triangleleft\ y$. Using the fact that $\sim$ is a simulation with respect to $\triangleright$, we conclude $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft\ y$.

2. $z \triangleleft y$: We have $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft \circ \triangleleft\ y$, and thus $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft\ y$.

3. $z \sim y$: We have $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft \circ \sim y$, thus using the fact that $\sim$ is a simulation with respect to $\triangleright$, we get $x \triangleright^* \circ \sim \circ\ {}^*\!\triangleleft\ y$. ■

Therefore, Theorem 14.1 follows directly from the following result:

Proposition 14.3. *For $\mathcal{R}_{\mathbb{T}}$ whose left-hand sides are linear and do not mention symbols in $\underline{\mathcal{F}}$, the relation $\simeq_{\mathcal{E}_{\mathbb{T}}}$ is a simulation with respect to $\longrightarrow_{\beta\underline{\beta}\mathcal{R}_{\mathbb{T}}}$. Diagrammatically,*

$$\begin{tikzcd}
u \arrow[r, phantom, "\simeq"] \arrow[d, dashed] & t \arrow[d] \\
\exists u' \arrow[r, phantom, "\simeq"] & t'
\end{tikzcd}$$

Proof. By induction on the rewrite context of $t \longrightarrow t'$. The crux of the proof is that, if $t = l[\theta]$ for some rewrite rule $l \longmapsto r \in \mathcal{R}_{\mathbb{T}}$, then linearity of l combined with the fact that it cannot mention symbols from $\underline{\mathcal{F}}$ imply that we must have $u = l[\theta']$ with $\theta \simeq \theta'$, and so $u = l[\theta'] \longrightarrow r[\theta'] \simeq r[\theta] = t'$. The same reasoning also discharges the cases $(x.v_1)v_2 \longrightarrow_{\beta} v_1[v_2/x]$ and $(\underline{x}.v)\underline{u} \longrightarrow_{\underline{\beta}} v[\underline{u}/\underline{x}]$. ■

Remark 14.1. Note that, if we had not syntactically separated confined terms from regular ones, the above proof would not have worked. For instance, taking $\mathcal{E}_{\mathbb{T}}$ with the equation $f(\mathsf{x}, \mathsf{x}) \approx g(\mathsf{x})$ and $\mathcal{R}_{\mathbb{T}}$ with $a \longmapsto b$, we would have

$$g(a) \simeq f(a, a) \longrightarrow f(b, a)$$

yet the only reduct of $g(a)$ is $g(b)$ and $g(b) \simeq f(b, a)$ does not hold. Therefore, it is crucial for our proof that regular terms can mention confined terms but not the other way around. □

Chapter 15

A Dedukti Theory for Predicative Universe Polymorphism

In this chapter we define $\mathbb{T}_{\mathbf{P}}^{\forall}$, a theory which extends $\mathbb{T}_{\mathbf{P}}$ (defined in Chapter 13) by internalizing universe levels and allowing for prenex universe polymorphism [HP91, ST14]. This is in particular a subtheory of the one implemented in the Agda proof assistant [Tea], and will be used as the new target of our proof translation. Before presenting it, let us mention that in previous work Genestier also proposed another Dedukti theory for predicative universe polymorphism [Gen20], yet, differently from our proposal, his one is not Church-Rosser modulo and furthermore requires the use of matching modulo associativity-commutativity, which is why we consider here a different theory.

15.1 Introducing $\mathbb{T}_{\mathbf{P}}^{\forall}$

The main change in the theory $\mathbb{T}_{\mathbf{P}}^{\forall}$ with respect to $\mathbb{T}_{\mathbf{P}}$ is that, instead of indexing symbols externally, we index them *inside* the framework [Ass15, Ste19]. To do this, we first introduce in Dedukti a syntax for *predicate universe levels*, by declaring confined symbols 0 (nullary), S (unary) and $\sqcup$ (binary).

Notation 15.1. In the following, we use the letter l to refer to confined terms built from 0, S and $\sqcup$, henceforth called *levels*, and we use the letters i, j, l to refer to confined variables occurring in them (though we will adopt more liberal notations in Chapter 17). To allow for more readable levels, we also adopt a lighter notation: we write $\sqcup$ in infix notation, S in curryfied notation, and consider $\sqcup$ as having a lower precedence than S — for instance, $\mathsf{S}\ i \sqcup j$ should be parsed as $\sqcup\ (\mathsf{S}(i), j)$. □

Universe levels are then subject to the following equational theory, which we call the *theory of predicative universe levels*, also used in the Agda proof assistant [Tea].[1]

$$\begin{array}{ccc} \mathtt{l}_1 \sqcup (\mathtt{l}_2 \sqcup \mathtt{l}_3) \approx (\mathtt{l}_1 \sqcup \mathtt{l}_2) \sqcup \mathtt{l}_3 & \mathrm{S}\,(\mathtt{l}_1 \sqcup \mathtt{l}_2) \approx \mathrm{S}\,\mathtt{l}_1 \sqcup \mathrm{S}\,\mathtt{l}_2 & \mathtt{l} \sqcup 0 \approx \mathtt{l} \\ \mathtt{l}_1 \sqcup \mathtt{l}_2 \approx \mathtt{l}_2 \sqcup \mathtt{l}_1 & \mathtt{l} \sqcup \mathrm{S}\,\mathtt{l} \approx \mathrm{S}\,\mathtt{l} & \mathtt{l} \sqcup \mathtt{l} \approx \mathtt{l} \end{array}$$

As we will see later in Chapter 17, not only these equations define a decidable theory, but they also ensure that two levels are convertible exactly when they are arithmetically equivalent, allowing us for instance to exchange $\mathrm{S}\ i \sqcup i \sqcup 0$ with $\mathrm{S}\ i$.

The declarations of Figure 5.2 defining PTSs in Dedukti are then replaced by the following ones, the only change being that the level parameters are now represented inside the framework. Note that the two rewrite rules are presented in linearized form, in order for them to be left-linear.

$$\begin{array}{l}
\mathrm{Ty} : \forall \mathtt{l}.\mathrm{Type} \\
\mathrm{Tm} : \forall \mathtt{l}.\mathrm{Ty}\ \mathtt{l} \to \mathrm{Type} \\
\mathrm{U} : \forall \mathtt{l}.\mathrm{Ty}\ (\mathrm{S}\ \mathtt{l}) \\
\mathrm{Tm}\ \mathtt{l}'\ (\mathrm{U}\ \mathtt{l}) \longmapsto \mathrm{Ty}\ \mathtt{l} \\
\Pi : \forall \mathtt{l}\mathtt{l}'.(\mathsf{A} : \mathrm{Ty}\ \mathtt{l}) \to (\mathsf{B} : \mathrm{Tm}\ \mathtt{l}\ \mathsf{A} \to \mathrm{Ty}\ \mathtt{l}') \to \mathrm{Ty}\ (\mathtt{l} \sqcup \mathtt{l}') \\
\lambda : \forall \mathtt{l}\mathtt{l}'.(\mathsf{A} : \mathrm{Ty}\ \mathtt{l}) \to (\mathsf{B} : \mathrm{Tm}\ \mathtt{l}\ \mathsf{A} \to \mathrm{Ty}\ \mathtt{l}') \to \\
\qquad ((x : \mathrm{Tm}\ \mathtt{l}\ \mathsf{A}) \to \mathrm{Tm}\ \mathtt{l}'\ (\mathsf{B}\ x)) \to \mathrm{Tm}\ (\mathtt{l} \sqcup \mathtt{l}')\ (\Pi\ \mathtt{l}\ \mathtt{l}'\ \mathsf{A}\ \mathsf{B}) \\
@ : \forall \mathtt{l}\mathtt{l}'.(\mathsf{A} : \mathrm{Ty}\ \mathtt{l}) \to (\mathsf{B} : \mathrm{Tm}\ \mathtt{l}\ \mathsf{A} \to \mathrm{Ty}\ \mathtt{l}') \to \\
\qquad (\mathsf{t} : \mathrm{Tm}\ (\mathtt{l} \sqcup \mathtt{l}')\ (\Pi\ \mathtt{l}\ \mathtt{l}'\ \mathsf{A}\ \mathsf{B})) \to (\mathsf{u} : \mathrm{Tm}\ \mathtt{l}\ \mathsf{A}) \to \mathrm{Tm}\ \mathtt{l}'\ (\mathsf{B}\ \mathsf{u}) \\
@\ \mathtt{l}\ \mathtt{l}'\ \mathsf{A}\ \mathsf{B}\ (\lambda\ \mathtt{l}''\ \mathtt{l}'''\ \mathsf{A}'\ \mathsf{B}'\ \mathsf{t})\ \mathsf{u} \longmapsto \mathsf{t}\ \mathsf{u}
\end{array}$$

This then concludes the definition of the theory $\mathbb{T}^{\forall}_{\mathbf{P}}$. In the following, we adopt a subscript notation for levels and write Ty_l, Tm_l, U_l, $\Pi_{l,l'}$, $\lambda_{l,l'}$ and $@_{l,l'}$ to improve clarity. When omitting arguments is mandatory for readability, we also continue to write $\Pi x : A.B$ for $\Pi_{l,l'}\ A\ (x.B)$ or $A \rightsquigarrow B$ when $x \notin \mathsf{fv}(B)$, and $\mathrm{Tm}\ A$ for $\mathrm{Tm}_l\ A$, and $\lambda x.t$ for $\lambda_{l,l'}\ A\ B\ (x.t)$, and $t_{@}u$ for $@_{l,l'}\ A\ B\ t\ u$.

Universe polymorphism in $\mathbb{T}^{\forall}_{\mathbf{P}}$ can be represented directly with the use of the framework's function type [Ass15]. Indeed, if a definition contains free level variables, it can be made universe polymorphic by abstracting over such variables. The following example illustrates this.

Example 15.1. The universe polymorphic identity function is given by

$$\begin{array}{l} \mathsf{id} : \forall i.\mathrm{Tm}\ (\Pi A : \mathrm{U}_i.A \rightsquigarrow A) \\ \mathsf{id} := i.\lambda A.\lambda a.a \end{array}$$

[1] Some authors consider a version of this theory without the neutral element 0 [BC22, BCDE23]; here we stick to the variant used in Agda, which includes the 0.

This then allows us to use id at any universe level: for instance, we can obtain the polymorphic identity function at the level 0 with the application id 0, which has the framework type $\mathrm{Tm}\ (\Pi A : \mathrm{U}_0 . A \rightsquigarrow A)$. □

Remark 15.1. Note that, unlike in Agda and the proposal of Bezem *et al.* [BCDE23], but similarly to the one of Sozeau and Tabareau [ST14], there is no object-level operation for universe level abstraction, which is instead handled by the framework function type. Therefore, universe polymorphic definitions are best understood as schemes rather than actual terms in the object logic. Alternatively, this can be understood as the fragment of Agda which only employs prenex level quantification. □

We conclude this short chapter by proving the following basic metaproperties of $\mathbb{T}^{\forall}_{\mathbf{P}}$.

Proposition 15.1 (Basic properties of $\mathbb{T}^{\forall}_{\mathbf{P}}$).

Well-typedness *The theory $\mathbb{T}^{\forall}_{\mathbf{P}}$ is well-typed.*

Church-Rosser Modulo *The rewrite system $\beta\underline{\beta}\mathcal{R}_{\mathbb{T}^{\forall}_{\mathbf{P}}}$ is Church-Rosser modulo $\mathcal{E}_{\mathbb{T}^{\forall}_{\mathbf{P}}}$.*

Subject reduction *If $\Gamma \vdash t : T$ and $t \longrightarrow t'$ then $\Gamma \vdash t' : T$.*

Proof. Well-typedness of $\mathbb{T}^{\forall}_{\mathbf{P}}$ can be easily verified manually. Church-Rosser modulo follows from Theorem 14.1 and the fact that $\beta\underline{\beta}\mathcal{R}_{\mathbb{T}^{\forall}_{\mathbf{P}}}$ is left-linear, confluent (by orthogonality [MN98]) and its left-hand sides do not mention 0, S or $\sqcup$. Finally, from Church-Rosser modulo we get the injectivity of $\longrightarrow$ and $\forall$, which yields subject reduction for β and $\underline{\beta}$ by Proposition 14.1, and using Church-Rosser modulo again we can also check that all rules in $\mathcal{R}_{\mathbb{T}^{\forall}_{\mathbf{P}}}$ preserve typing, which yields subject reduction for $\mathcal{R}_{\mathbb{T}^{\forall}_{\mathbf{P}}}$ by Proposition 14.1. ■

Chapter 16

Elaborating Universe-Polymorphic Definitions

Now that we have refined the target of our translation from $\mathbb{T}_{\mathbf{P}}$ to $\mathbb{T}_{\mathbf{P}}^{\forall}$, we update the statement of the problem we are trying to solve: given a local signature Φ in the theory $\mathbb{T}_{\mathbf{I}}$ — or actually in any theory given by instantiating Figure 5.2 with a PTS specification — we want to translate it to a local signature Φ' well-typed in the theory $\mathbb{T}_{\mathbf{P}}^{\forall}$. As anticipated in the end of Chapter 13, we propose to do this by incrementally elaborating each entry of Φ into a universe-polymorphic one in $\mathbb{T}_{\mathbf{P}}^{\forall}$, allowing for each previously translated entry to be usable at various levels. Of course, the elaboration might fail at some point, however if all steps succeed then we should get a local signature Φ' well-typed in $\mathbb{T}_{\mathbf{P}}^{\forall}$.

16.1 A bidirectional elaborator

In order to explain how a local signature Φ can be translated, we first explain how a single term can be elaborated into a universe-polymorphic term in $\mathbb{T}_{\mathbf{P}}^{\forall}$. This is done by adapting the seminal work of Harper and Pollack [HP91], which defines a type system for elaborating terms written using the *typical ambiguity* discipline, where universe levels are left implicit. Compared with their work, the main differences is that we target a different theory, and that our elaborator is bidirectional, similarly to the one of Norell [Nor07].

Recall from Part II that in bidirectional type systems the typing judgment $\Gamma \vdash t : T$ is split into modes infer $\Gamma \vdash t \Rightarrow T$ and check $\Gamma \vdash t \Leftarrow T$. In mode infer we start with Γ, t and we output a type T for t in Γ, whereas in mode check we are given Γ, t, T and we check that t indeed has type T in Γ. Crucial in bidirectional typing is the proper bookkeeping of pre-conditions and post-conditions, which are resumed in the following table — there, we mark inputs with $-$ and outputs with $+$.

Judgment	Pre-condition	Post-condition
$\Gamma^- \vdash t^- \Rightarrow T^+$	$\Gamma \vdash$	$\Gamma \vdash t : T$
$\Gamma^- \vdash t^- \Leftarrow T^-$	$\Gamma \vdash T : s$	$\Gamma \vdash t : T$

To define our elaborator, we refine the standard bidirectional judgments in two ways: first by outputting the term resulting from the elaboration, and second by returning a set of equational constraints under which the term is well-typed. In order to do the latter, let us start with some preliminary definitions. A *level unification problem* C is a set containing equations of the form $l \overset{?}{=} l'$, sometimes also referred to as *constraints*. In the following definitions, let θ be a *level substitution*, meaning that $i[\theta]$ is a level for all $i \in \mathrm{dom}(\theta)$.[1] We write $\theta \vDash C$ when $l[\theta] \simeq l'[\theta]$ for all $l \overset{?}{=} l' \in C$, in which case θ is called a *unifier* (or *solution*) for C. We then write $\Gamma \vdash_C$ when $\theta \vDash C$ implies $\Gamma[\theta] \vdash$ for all θ, and $\Gamma \vdash_C t : T$ when $\theta \vDash C$ implies $\Gamma[\theta] \vdash t[\theta] : T[\theta]$ for all θ.

Intuitively, just like the context Γ in $\Gamma \vdash t : T$ allows us to state a typing judgment $t : T$ with typing hypotheses of the form $x : B \in \Gamma$, the set C in $\Gamma \vdash_C t : T$ refines this with equational hypotheses of the form $l \simeq l'$. With this in mind, we can now give the new typing judgments in the following table. Compared with the previous table, we now start with a set of constraints $\mathcal{D}$ that guarantees that the pre-condition holds, and in the process we output a term t' and a set of constraints C that, together with $\mathcal{D}$, ensure that the returned term has the expected type.

Judgment	Pre-condition	Post-condition
$\Gamma^- \uparrow \mathcal{D}^- \vdash t^- \Rightarrow T^+ \downarrow C^+ \rightsquigarrow t'^+$	$\Gamma \vdash_{\mathcal{D}}$	$\Gamma \vdash_{C \cup \mathcal{D}} t' : T$
$\Gamma^- \uparrow \mathcal{D}^- \vdash t^- \Leftarrow T^- \downarrow C^+ \rightsquigarrow t'^+$	$\Gamma \vdash_{\mathcal{D}} T : s$	$\Gamma \vdash_{C \cup \mathcal{D}} t' : T$

We can now define the bidirectional elaborator by the rules in Figure 16.1. In rule Sym we write $\hat{(-)}$ for the function mapping the symbols $\mathrm{Ty}_l, \mathrm{Tm}_l, \mathrm{U}_l, \Pi_{l,l'}, @_{l,l'}, \lambda_{l,l'}$ from Figure 5.2 to the symbols $\mathrm{Ty}, \mathrm{Tm}, \mathrm{U}, \Pi, @, \lambda$ from Chapter 15, and keeping any other symbol unchanged.

The elaborator also relies on two new conversion judgments $T \equiv U \downarrow C$ and $T \equiv^{\mathsf{h}} U \downarrow C$ used to compute a set of constraints needed for the conversion to hold. In the given rules, we write $t \longrightarrow^{\mathsf{h}} u$ for the reduction of t to a *head-normal form* u, meaning that $t \longrightarrow^{*} u$ and, if $u \longrightarrow^{*} u'\, v_1 \ldots v_k$ (where k might be 0), then u' matches no rewrite rule left-hand side at the head (including β and $\underline{\beta}$).[2]

Remark 16.1. Note that the elaborator of Figure 16.1 is not defined for the whole syntax of the framework — for instance, it is not defined for t not in β-normal form. This is because we are only interested in the fragment of $\mathbb{T}_{\mathbf{I}}$ which is in the image of the translation function defined in Chapter 5. □

[1] That is, θ may only mention confined symbols declared in the theory $\mathbb{T}^{\forall}_{\mathbf{P}}$.

[2] Therefore, the definition of head-normal form is *not* the same one as in Section 8.2, which would allow for instance for terms like $(x.x)yz$. This is because Dedukti uses a different formalism of higher-order rewriting, in which the definition of head-normal form is instead adapted from the λ-calculus.

$$\boxed{t \equiv u \downarrow C}$$

$$\frac{t \longrightarrow^{\mathsf{h}} t' \qquad u \longrightarrow^{\mathsf{h}} u' \qquad t' \equiv^{\mathsf{h}} u' \downarrow C}{t \equiv u \downarrow C}$$

$$\boxed{t \equiv^{\mathsf{h}} u \downarrow C}$$

$$\frac{t = x, f, s}{t \equiv^{\mathsf{h}} t \downarrow \emptyset} \qquad \frac{t \equiv t' \downarrow C}{x.t \equiv^{\mathsf{h}} x.t' \downarrow C} \qquad \frac{t \equiv^{\mathsf{h}} t' \downarrow C}{t\ l \equiv^{\mathsf{h}} t'\ l' \downarrow C \cup \{l \overset{?}{=} l'\}}$$

$$\frac{T \equiv T' \downarrow C_1 \qquad U \equiv U' \downarrow C_2}{(x : T) \to U \equiv^{\mathsf{h}} (x : T') \to U' \downarrow C_1 \cup C_2} \qquad \frac{t \equiv^{\mathsf{h}} t' \downarrow C_1 \qquad u \equiv u' \downarrow C_2}{t\ u \equiv^{\mathsf{h}} t'\ u' \downarrow C_1 \cup C_2}$$

$$\boxed{\Gamma \uparrow \mathcal{D} \vdash t \Leftarrow T \downarrow C \rightsquigarrow t'}$$

$$\textsc{Switch}\ \frac{\Gamma \uparrow \mathcal{D} \vdash t \Rightarrow T \downarrow C_1 \rightsquigarrow t' \qquad T \equiv U \downarrow C_2}{\Gamma \uparrow \mathcal{D} \vdash t \Leftarrow U \downarrow C_1 \cup C_2 \rightsquigarrow t'} \qquad \textsc{Abs}\ \frac{\begin{array}{c} T \longrightarrow^{\mathsf{h}} (x : T_1) \to T_2 \\ \Gamma, x : T_1 \uparrow \mathcal{D} \vdash t \Leftarrow T_2 \downarrow C \rightsquigarrow t' \end{array}}{\Gamma \uparrow \mathcal{D} \vdash x.t \Leftarrow T \downarrow C \rightsquigarrow x.t'}$$

$$\boxed{\Gamma \uparrow \mathcal{D} \vdash t \Rightarrow T \downarrow C \rightsquigarrow t'}$$

$$\textsc{Sym}\ \frac{\hat{f} : \forall j_1 \ldots j_k . T \in \mathbb{T} \qquad i_1, \ldots, i_k \text{ fresh}}{\Gamma \uparrow \mathcal{D} \vdash f \Rightarrow T[\vec{i}/\vec{j}] \downarrow \emptyset \rightsquigarrow \hat{f}\ i_1 \ldots i_k} \qquad \textsc{Var}\ \frac{x : T \in \Gamma}{\Gamma \uparrow \mathcal{D} \vdash x \Rightarrow T \downarrow \emptyset \rightsquigarrow x}$$

$$\textsc{App}\ \frac{\Gamma \uparrow \mathcal{D} \vdash t \Rightarrow T \downarrow C_1 \rightsquigarrow t' \qquad T \longrightarrow^{\mathsf{h}} (x : T_1) \to T_2 \qquad \Gamma \uparrow \mathcal{D} \cup C_1 \vdash u \Leftarrow T_1 \downarrow C_2 \rightsquigarrow u'}{\Gamma \uparrow \mathcal{D} \vdash t\ u \Rightarrow T_2[u'/x] \downarrow C_1 \cup C_2 \rightsquigarrow t'\ u'}$$

Figure 16.1: Bidirectional type system for universe-polymorphic elaboration

Example 16.1. Suppose we want to elaborate the entry

$$\begin{aligned}&\mathsf{id}' : \mathrm{Tm}\ (\Pi A : \mathrm{U}_\Omega.A \rightsquigarrow A)\\ &\mathsf{id}' := \mathsf{id}_{@}(\Pi A : \mathrm{U}_\Omega.A \rightsquigarrow A)_{@}\mathsf{id}\end{aligned}$$

in an extension of $\mathbb{T}^{\forall}_{\mathbf{P}}$ with $\mathsf{id} : \forall i.(\Pi A : \mathrm{U}_i.A \rightsquigarrow A) := i.\lambda A.\lambda x.x$. To do so, we first elaborate $\mathrm{Tm}\ (\Pi A : \mathrm{U}_\Omega.A \rightsquigarrow A)$ into a valid type, yielding

$$() \uparrow \emptyset \vdash \mathrm{Tm}\ (\Pi A : \mathrm{U}_\Omega.A \rightsquigarrow A) \Leftarrow \mathsf{Type} \downarrow C_1 \rightsquigarrow \mathrm{Tm}\ (\Pi A : \mathrm{U}_{i_1}.A \rightsquigarrow A)$$

We can then elaborate $\mathsf{id}_{@}(\Pi A : \mathrm{U}_\Omega.A \rightsquigarrow A)_{@}\mathsf{id}$ under the constraints C_1:

$$\begin{aligned}() \uparrow C_1 \vdash\ &\mathsf{id}_{@}(\Pi A : \mathrm{U}_\Omega.A \rightsquigarrow A)_{@}\mathsf{id} \Leftarrow \mathrm{Tm}\ (\Pi A : \mathrm{U}_{i_1}.A \rightsquigarrow A) \downarrow C_2\\ &\rightsquigarrow (\mathsf{id}\ i_2)_{@}(\Pi A : \mathrm{U}_{i_3}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4)\end{aligned}$$

In the end we get the constraints

$$C_1 \cup C_2 := \{\mathrm{S}\ i_1 \overset{?}{=} i_2, i_1 \overset{?}{=} i_3, i_1 \overset{?}{=} i_4, \ldots\}$$

where the hidden constraints concern level variables appearing in implicit arguments. □

We can show the soundness of elaboration by simply verifying that each rule locally preserves the invariants of the last table — this is the essence of the proof of Theorem 16.1. Before proving this, we first need a lemma establishing the soundness of conversion checking.

Lemma 16.1. *Suppose that $T \equiv U \downarrow C$ or $T \equiv^{\mathsf{h}} U \downarrow C$. If $\theta \vDash C$ then $T[\theta] \equiv U[\theta]$.*

Proof. By an easy mutual induction on $T \equiv U \downarrow C$ and $T \equiv^{\mathsf{h}} U \downarrow C$. ■

Theorem 16.1 (Soundness of term elaboration). *In the following, suppose that the underlying theory satisfies well-typedness and subject reduction.*

- *If $\Gamma \vdash_{\mathcal{D}} T : s$ and $\Gamma \uparrow \mathcal{D} \vdash t \Leftarrow T \downarrow C \rightsquigarrow t'$ then $\Gamma \vdash_{C \cup \mathcal{D}} t' : T$*
- *If $\Gamma \vdash_{\mathcal{D}}$ and $\Gamma \uparrow \mathcal{D} \vdash t \Rightarrow T \downarrow C \rightsquigarrow t'$ then $\Gamma \vdash_{C \cup \mathcal{D}} t' : T$*

Proof. By mutual induction on the elaborator judgments.

- Case Var. Let $\theta \vDash \emptyset \cup \mathcal{D}$. By hypothesis we have $\Gamma[\theta] \vdash$, and hence $\Gamma[\theta] \vdash x : T[\theta]$.
- Case Sym. Let $\theta \vDash \emptyset \cup \mathcal{D}$. By hypothesis we have $\Gamma[\theta] \vdash$, and hence $\Gamma[\theta] \vdash \hat{f} : \forall \vec{j}.T$. Because θ is a level substitution, then $i_1[\theta], \ldots, i_k[\theta]$ are all levels, so we can derive $\Gamma[\theta] \vdash f\ \vec{i}[\theta] : T[\vec{i}[\theta]/\vec{j}]$. Finally, θ is not defined for $\vec{j}$, so $T[\vec{i}[\theta]/\vec{j}] = T[\vec{i}/\vec{j}][\theta]$.

- Case Switch. Let $\theta \vDash \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}$. By hypothesis we have $\Gamma \vdash_{\mathcal{D}} U : s$ and thus $\Gamma \vdash_{\mathcal{D}}$, therefore by i.h. we have $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} t' : T$, from which we get $\Gamma[\theta] \vdash t'[\theta] : T[\theta]$. Moreover, from Lemma 16.1 we also get $T[\theta] \equiv U[\theta]$. Finally, from $\Gamma \vdash_{\mathcal{D}} U : s$ we have $\Gamma[\theta] \vdash U[\theta] : s$, and therefore we conclude $\Gamma[\theta] \vdash t'[\theta] : U[\theta]$ by conversion.

- Case App. By hypothesis we have $\Gamma \vdash_{\mathcal{D}}$, therefore by i.h. we have $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} t' : T$.

  We first claim that $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} T_1 : \mathsf{Type}$ and $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} t' : (x : T_1) \to T_2$. Let $\theta \vDash \mathcal{C}_1 \cup \mathcal{D}$, in which case we have $\Gamma[\theta] \vdash t'[\theta] : T[\theta]$. By validity we have $\Gamma[\theta] \vdash T[\theta] : s$ for some s, so by subject reduction and $T[\theta] \longrightarrow^* (x : T_1[\theta]) \to T_2[\theta]$ we have $\Gamma[\theta] \vdash (x : T_1[\theta]) \to T_2[\theta] : s$. By inversion we thus get $\Gamma[\theta] \vdash T_1[\theta] : \mathsf{Type}$. Finally, applying conversion with $\Gamma[\theta] \vdash t'[\theta] : T[\theta]$, we get $\Gamma[\theta] \vdash t'[\theta] : (x : T_1[\theta]) \to T_2[\theta]$.

  Using $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{D}} T_1 : \mathsf{Type}$, we can now apply the i.h. again and obtain $\Gamma \vdash_{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}} u' : T_1$. Now let $\theta \vDash \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{D}$. We thus have $\Gamma[\theta] \vdash t'[\theta] : (x : T_1[\theta]) \to T_2[\theta]$ and $\Gamma[\theta] \vdash u'[\theta] : T_1[\theta]$, so by the application rule we get $\Gamma[\theta] \vdash t'[\theta]\ u'[\theta] : T_2[\theta][u'[\theta]/x]$. Because $T_2[\theta][u'[\theta]/x] = T_2[u'/x][\theta]$, the result follows.

- Case Abs. By hypothesis we have $\Gamma \vdash_{\mathcal{D}} T : s$, from which we can easily derive $\Gamma \vdash_{\mathcal{D}} T_1 : \mathsf{Type}$ and $\Gamma, x : T_1 \vdash_{\mathcal{D}} T_2 : s$, like in the previous case.

  Now let $\theta \vDash \mathcal{C} \cup \mathcal{D}$. By the i.h. we get also get $\Gamma, x : T_1 \vdash_{\mathcal{C} \cup \mathcal{D}} t' : T_2$, and thus $\Gamma[\theta], x : T_1[\theta] \vdash t'[\theta] : T_2[\theta]$, and from $\Gamma \vdash_{\mathcal{D}} T_1 : \mathsf{Type}$ and $\Gamma, x : T_1 \vdash_{\mathcal{D}} T_2 : s$ we also get $\Gamma[\theta] \vdash T_1[\theta] : \mathsf{Type}$ and $\Gamma[\theta], x : T_1[\theta] \vdash T_2[\theta] : s$. Therefore, we conclude $\Gamma[\theta] \vdash x.t'[\theta] : (x : T_1[\theta]) \to T_2[\theta]$ by the abstraction rule. ■

16.2 Elaborating local signatures

Using the term elaborator of last section, we can then translate a local signature Φ by simply elaborating each of its entries, in the theory given by $\mathbb{T}^{\forall}_{\mathbf{P}}$ extended with the prefix preceding the entry in question.

However, because the result of elaborating each entry generates a set of constraints, in order to obtain a well-typed term we first need to solve them. Yet, as explained in Chapter 13, we do not want a numerical assignment of level variables that satisfies the constraints, but rather a general symbolic solution which allows the term to be instantiated later at different universe levels. This, therefore, requires the use of unification. However, because levels are not purely syntactic entities, one needs to devise a unification algorithm specific for the equational theory of universe levels, which will be the purpose of the next chapter. So for now, let us just assume we are given a (partial) function Unify which computes from a set of constraints $\mathcal{C}$ a substitution θ with $\theta \vDash \mathcal{C}$.

The elaboration of a local signature can then be defined by the rules in Figure 16.2. Let us explain the case of elaborating an entry $f : T := t$. The first step is elaborating T and

$\boxed{\Phi \rightsquigarrow \Phi'}$

$$\frac{}{\cdot \rightsquigarrow \cdot} \qquad \frac{\Phi \rightsquigarrow \Phi' \qquad \mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \uparrow \emptyset \vdash T \Leftarrow \mathsf{Type} \downarrow C \rightsquigarrow T' \\ \theta = \textsc{Unify}(C) \qquad \vec{i} = \mathsf{fv}(T'[\theta])}{\Phi, f : T \rightsquigarrow \Phi', f : \forall \vec{i}.T'[\theta]}$$

$$\frac{\Phi \rightsquigarrow \Phi' \qquad \mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \uparrow \emptyset \vdash T \Leftarrow \mathsf{Type} \downarrow C \rightsquigarrow T' \\ \mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \uparrow C \vdash t \Leftarrow T' \downarrow \mathcal{D} \rightsquigarrow t' \\ \theta = \textsc{Unify}(C \cup \mathcal{D}) \qquad \vec{i} = \mathsf{fv}(T'[\theta]) \qquad \vec{j} = \mathsf{fv}(t'[\theta]) \setminus \mathsf{fv}(T'[\theta])}{\Phi, f : T := t \rightsquigarrow \Phi', f : \forall \vec{i}.T'[\theta] := \vec{i}.t'[\theta][0/\vec{j}]}$$

Figure 16.2: Local signature elaboration

then t, yielding the terms t', T' and the constraints C and $\mathcal{D}$, and then calculating a unifier for them. If we write $\vec{i}$ for $\mathsf{fv}(T'[\theta])$ and $\vec{j}$ for $\mathsf{fv}(t'[\theta]) \setminus \mathsf{fv}(T'[\theta])$, we could then simply generalize over $\vec{i}, \vec{j}$ and obtain the entry $f : \forall \vec{i}\vec{j}.T'[\theta] := \vec{i}\vec{j}.t[\theta']$. However, because the $\vec{j}$ do not appear on the type $T'[\theta]$, then in order to reduce the number of level arguments we can simply map them to the bottom level 0, yielding $f : \forall \vec{i}.T'[\theta] := \vec{i}.t[\theta'][0/\vec{j}]$ instead. Even though this does not impact the soundness of the elaboration, this optimization is still useful because reducing the number of level arguments empirically leads to unification problems that are easier to solve in practice.

Example 16.2. Suppose that we want to translate the last example of Chapter 13, namely

$$\begin{aligned} \Phi = \ & \mathsf{id} : (\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A) := \lambda A.\lambda t.t, \\ & \mathsf{id}' : \mathsf{Tm}\ (\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A) := \mathsf{id}_{@}(\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A)_{@}\mathsf{id} \end{aligned}$$

For the sake of this example, we skip the calculation of the first part of Φ, which yields

$$\mathsf{id} : (\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A) := \lambda A.\lambda t.t \quad \rightsquigarrow \quad \mathsf{id} : \forall i.(\Pi A : \mathrm{U}_{i}.A \rightsquigarrow A) := i.\lambda A.\lambda t.t$$

Now recall that, by Example 16.1, elaborating the terms

$$\mathsf{Tm}\ (\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A) \quad \text{and} \quad \mathsf{id}_{@}(\Pi A : \mathrm{U}_{\Omega}.A \rightsquigarrow A)_{@}\mathsf{id}$$

yields the terms

$$\forall i.\mathsf{Tm}\ (\Pi A : \mathrm{U}_{i}.A \rightsquigarrow A) \quad \text{and} \quad (\mathsf{id}\ i_2)_{@}(\Pi A : \mathrm{U}_{i_3}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4)$$

and the equations

$$C_1 \cup C_2 := \{\mathrm{S}\ i_1 \overset{?}{=} i_2, i_1 \overset{?}{=} i_3, i_1 \overset{?}{=} i_4, \ldots\}$$

The algorithm of the next section is able to compute the unifier

$$\theta = i_1 \mapsto i_4,\ i_2 \mapsto \mathsf{S}\ i_4,\ i_3 \mapsto i_4, \dots$$

so we can now apply the unifier to the generated terms and generalize over the free level variables of the type, while mapping the other ones to 0, giving at the end

$$\mathsf{id}' : \forall i_4.\mathsf{Tm}\ (\Pi A : \mathsf{U}_{i_4}.A \rightsquigarrow A) := i_4.(\mathsf{id}\ (\mathsf{S}\ i_4))_{@}(\Pi A : \mathsf{U}_{i_4}.A \rightsquigarrow A)_{@}(\mathsf{id}\ i_4)$$

Note that in the resulting term, the two occurrences of id are applied to different universe levels, illustrating the importance of using universe polymorphism in the translation. □

Let us now show the final soundness theorem for our translation. For this, we will need the following easy lemma.

Lemma 16.2. *If Φ is well-typed in $\mathbb{T}^{\forall}_{\mathbf{P}}$ then $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi$ satisfies well-typedness and subject reduction.*

Proof. Well-typedness of $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi$ is a trivial consequence of well-typedness of $\mathbb{T}^{\forall}_{\mathbf{P}}$ (Proposition 15.1) and $\mathbb{T}^{\forall}_{\mathbf{P}} \vdash \Phi$. To show subject reduction, we will also need Church-Rosser modulo, which follows by Theorem 14.1, using the fact that $\mathcal{R}_{\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi}$ is orthogonal and hence confluent. Then, subject reduction for $\beta\underline{\beta}$ follows from injectivity of $\longrightarrow$ and $\forall$ (which we have by Church-Rosser modulo), and subject reduction for the other rewrite rules follow by verifying that they all preserve typing. The preservation of typing of the rules already in $\mathbb{T}^{\forall}_{\mathbf{P}}$ is shown by the same proof as in Proposition 15.1, and for the other rules $f \longmapsto t$ we use $\mathbb{T}^{\forall}_{\mathbf{P}} \vdash \Phi$ to obtain a derivation of $\cdot \vdash t : T$ for each $f : T := t \in \Phi$. ■

Theorem 16.2 (Soundness of local signature elaboration). *If $\Phi \rightsquigarrow \Phi'$ then $\mathbb{T}^{\forall}_{\mathbf{P}} \vdash \Phi'$.*

Proof. By induction on the definition of the judgment $\Phi \rightsquigarrow \Phi'$, the base case being trivial. Let us consider the case of a local signature $\Phi, f : T := t$, the case $\Phi, f : T$ being similar. By applying the i.h. to $\Phi \rightsquigarrow \Phi'$, we obtain that Φ' is well-typed in $\mathbb{T}^{\forall}_{\mathbf{P}}$, implying by Lemma 16.2 that $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi'$ is well-typed and satisfies subject reduction. Therefore, we can apply Theorem 16.1 to get $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \vdash_{\mathcal{D}} T' : \mathsf{Type}$ and $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \vdash_{C \cup \mathcal{D}} t' : T'$, and because θ is a unifier for $C \cup \mathcal{D}$, we get $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \vdash t'[\theta] : T'[\theta]$. By the substitution property, and the fact that the $\vec{j}$ do not occur in $T'[\theta]$, we obtain $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \vdash t'[\theta][0/\vec{j}] : T'[\theta]$, and thus $\mathbb{T}^{\forall}_{\mathbf{P}} \ltimes \Phi' \triangleright \cdot \vdash \vec{i}.t'[\theta][0/\vec{j}] : \forall\vec{i}.T'[\theta][0/\vec{j}]$, allowing us to conclude $\mathbb{T}^{\forall}_{\mathbf{P}} \vdash \Phi', f : \forall\vec{i}.T'[\theta] := \vec{i}.t'[\theta][0/\vec{j}]$. ■

Chapter 17

Unification for Predicative Universe Levels

The elaborator presented in the last chapter relies on an unspecified algorithm for universe level unification, which we now define in this chapter.

Notation 17.1. We adopt new notation conventions for the rest of this chapter. First, we write $t, u, v, \ldots$ for levels, instead of l, l', and $x, y, \ldots$ for level variables, instead of i, j[1] — because the only terms we deal with in this chapter are levels (except for the proof of Theorem 17.2, where we return to the previous notations), no ambiguity can arise. Moreover, to improve the readability of large level expressions, we write $n + t$ for the level $\mathsf{S}^n\ t$, n for the level $\mathsf{S}^n\ 0$, and we drop the blue color in $\sqcup$. For instance, the level $\mathsf{S}\ 0\ \sqcup\ \mathsf{S}\ (\mathsf{S}\ x\ \sqcup\ 0)$ will henceforth be written $1 \sqcup 1 + (1 + x \sqcup 0)$ — note that $+$ binds tighter then $\sqcup$, and that the left argument of $+$ is always a natural number, so this expression can be parsed unambiguously. Finally, we also allow ourselves to write entries of substitutions θ as $x \mapsto t$ instead of t/x, which sometimes can improve readability when t is a big term. □

Preliminaries about unification

Before going any further, let us review some important preliminaries about unification theory — we refer to Baader and Snyder [BS01] for a thorough introduction to this topic. Recall from the previous section that we consider unification problems C containing equations of the form $t \overset{?}{=} u$, for t and u levels, and that a unifier θ for C is a level substitution for which we have $t[\theta] \simeq u[\theta]$ for all equations $t \overset{?}{=} u$ in C, in which case we write $\theta \vDash C$.[2]

Given a finite set X of level variables, we define the *instantiation pre-order* $\preceq^X$ by $\theta \preceq^X \tau$ iff there is some σ such that $x[\theta][\sigma] \simeq x[\tau]$ for all $x \in X$. We then say that a set Θ of unifiers for C is *complete* if, for any unifier τ of C, we have $\theta \preceq^{\mathsf{fv}(C)} \tau$ for some $\theta \in \Theta$,

[1] This frees i, j so we can use them for indexing, as is common in many mathematical texts.

[2] Therefore, in this thesis we only consider the problem of *elementary* unification, in which only symbols from the equational theory and variables might appear in the problem or the solutions.

and we say that it is *minimal* if any two unifiers $\theta, \theta' \in \Theta$ are incomparable with respect to $\leq^{\mathsf{fv}(C)}$. We often abbreviate "minimal complete set of unifiers" as just m.c.u..

Because m.c.u.s are unique up to the equivalence relation defined by $\leq^{\mathsf{fv}(C)}$ [BS01, Corollary 3.13], it is possible to classify unification problems with respect to the cardinality of their m.c.u.s. Given a unification problem C, we then say that C is *unitary* when its m.c.u.s are of cardinality at most 1,[3] *finitary* if they are of finite cardinality, *infinitary* if they are of infinite cardinality, and *nullary* if m.c.u.s do not exist. By ordering these as unitary < finitary < infinitary < nullary, we then say that an equational theory is unitary (resp. finitary, infinitary, nullary) when its unification problems are at most unitary (resp. finitary, infinitary, nullary). Finally, when a problem is unitary, the elements of its singleton m.c.u.s are called *most general unifiers*, and often abbreviated as just m.g.u.s.

The theory of predicative universe levels is not unitary

When studying unification in an equational theory, a very natural first question is whether the theory is unitary, given that unitary theories are much more convenient to handle from the point of view of unification theory. Unfortunately, our first observation is that the equational theory of predicative levels used in $\mathbb{T}^{\forall}_{\mathbf{P}}$ is not unitary.

Theorem 17.1. *The theory of predicative universe levels is not unitary, that is, there is some solvable unification problem that has no most general unifier.*

Proof. Consider the equation $1 + x_1 \overset{?}{=} x_2 \sqcup x_3$, which is solvable, and suppose it had a m.g.u. θ. Note that $\theta_1 = x_1 \mapsto 0,\ x_2 \mapsto 1,\ x_3 \mapsto 0$ is also a unifier, thus for some τ we have $x_3[\theta][\tau] \simeq 0$. Therefore, there can be no occurrence of a successor in $x_3[\theta]$. By taking $\theta_2 = x_1 \mapsto 0,\ x_2 \mapsto 0,\ x_3 \mapsto 1$ we can show similarly that there can be no occurrence of a successor in $x_2[\theta]$. But by taking a substitution θ' mapping all variables in $(x_2 \sqcup x_3)[\theta]$ to 0, we get $(x_2 \sqcup x_3)[\theta][\theta'] \simeq 0$, which cannot be equivalent to $(1 + x_1)[\theta][\theta']$. Hence, $1 + x_1 \overset{?}{=} x_2 \sqcup x_3$ has no m.g.u. ∎

One can then also wonder if, by restricting to the fragment of problems generated by the elaborator, one can expect to recover the property that all solvable problems admit a m.g.u. The following result also answers this negatively.

Theorem 17.2. *There is a local signature whose constraints computed by the elaborator are solvable but have no most general unifier.*

Proof. In this proof, we locally return to the conventions of Notation 15.1. Consider the following singleton signature, which is well-typed in $\mathbb{T}_{\mathbf{P}}$. Once again, we reuse our convention of keeping some arguments implicit.

$$\begin{aligned} \Phi = \quad & f : \mathsf{Tm}\ (\mathsf{U}_1 \rightsquigarrow \mathsf{U}_1 \rightsquigarrow (\Pi C : \mathsf{U}_2 . C \rightsquigarrow C \rightsquigarrow \mathsf{U}_0) \rightsquigarrow \mathsf{U}_0) \\ & f := \lambda A.\lambda B.\lambda R.R_{@}\mathsf{U}_{1@}\mathsf{U}_{0@}(A \rightsquigarrow B) \end{aligned}$$

[3] Note therefore that a problem that has no unifier is automatically unitary.

Its elaboration yields the terms

$$\mathrm{Tm}\ (\mathrm{U}_{i_1} \rightsquigarrow \mathrm{U}_{i_2} \rightsquigarrow (\Pi C : \mathrm{U}_{i_3}.C \rightsquigarrow C \rightsquigarrow \mathrm{U}_{i_4}) \rightsquigarrow \mathrm{U}_{i_5})$$

and

$$\lambda A.\lambda B.\lambda R.R_{@}\mathrm{U}_{i_6}{}_{@}\mathrm{U}_{i_7}{}_{@}(A \rightsquigarrow B)$$

and a unification problem that can be simplified to $\{\mathrm{S}\ i_7 \overset{?}{=} i_1 \sqcup i_2\}$, after solving some easy equations.[4] This is because the application of $R_{@}\mathrm{U}_{i_6} : \mathrm{Tm}\ (\mathrm{U}_{i_6} \rightsquigarrow \mathrm{U}_{i_6} \rightsquigarrow \mathrm{U}_{i_4})$ to U_{i_7} and $(A \rightsquigarrow B)$ requires the last two to be in the same universe, whose levels are respectively $\mathrm{S}\ i_7$ and $i_1 \sqcup i_2$. This equation is solvable but, by Theorem 17.1, does not admit a most general unifier. ■

Remark 17.1. Note that the definition of the above proof can be typed predicatively, showing that impredicativity is not the culprit of the problem. One can also wonder if a similar term can be typed in $\mathbb{T}_{\mathbf{I}}$. It seems that the idea of the above counter-example requires at least 3 universe levels while $\mathbb{T}_{\mathbf{I}}$ has only 2. However, impredicative proof assistants such as Coq, Matita and Lean often consider an infinite hierarchy of universes, in which the counter-example can be reproduced. □

Remark 17.2. In the above proof, one can alternatively verify the calculation of constraints automatically in Agda by typechecking the code

```
test : (A : Set _) → (B : Set _) → (R : (C : Set _) → C → C → Set _) → Set _
test = λA B R → R (Set _) (Set _) (A → B)
```

which returns the error

```
Failed to solve the following constraints: _0 ⊔ _1 = lsuc _10
```

showing that Agda's elaborator also simplifies the problem to find the same constraint. □

In other words, some elaborated terms may not admit a most general universe-polymorphic instance, even when they admit some well-typed instances. A possible strategy would be to look not for a m.g.u., but instead for a minimal set of incomparable unifiers, as is often done in the equational unification literature [BS01]. However, this would not only require to duplicate each term being translated, one for each incomparable unifier, but this strategy would also risk of growing the output size exponentially. Indeed, a term using a previous translated entry that was duplicated n times would then need to be elaborated multiple times, once with each of these n variants. Therefore, we at first insist in looking only for m.g.u.s, even if by Theorem 17.2 this approach can fail in some cases when the problem is solvable but does not admit a m.g.u.

[4] For instance, by using the algorithm of Figure 17.2.

Overview of the chapter

The first main contribution of this chapter, given in Section 17.2, is a complete characterization of the equations $t \stackrel{?}{=} u$ that admit a most general unifier. More precisely, our result says exactly when such a single equation (1) admits a m.g.u., in which case we also have an explicit description of one, (2) does not admit any unifier, or (3) admits some unifier but no most general one. However, because we are interested in unification problems that may contain multiple equations, in Section 17.3 we then apply this characterization in the design of an algorithm using a *constraint-postponing* strategy [ZS17, DHKP96, Ree09]: at each step, we look for an equation which admits a m.g.u. and eliminate it, while applying the obtained substitution to the other constraints. This can then bring new equations to the fragment admitting a m.g.u., allowing us to solve them next. This is similar to how most proof assistants handle higher-order unification problems, by trying to solve the equations that are in the *pattern* fragment, in the hope of unblocking some other ones in the process. When applying our algorithm to a problem, there are then three possible outcomes: (1) it yields a substitution, in which case it is a m.g.u., (2) it yields $\bot$, in which case there are no unifiers, or (3) it gets stuck in the process and produces no solution.

Our algorithm is based on a complete characterization of single equations, so it follows that it is complete for finding m.g.u.s when the problem has only one equation. Yet, outside of this class, there are actually problems with m.g.u.s for which our algorithm does not return any unifier. We then contribute a second algorithm in Section 17.4 which we show to calculate a minimal complete set of unifiers under two conjectures about linear problems in max-plus algebra — in particular, under these hypotheses, it is able to find a m.g.u. for all problems that admit one. More explicitly, our algorithm works similarly to Baader's unification algorithm for commutative theories [BS01] — though we stress that the equational theory of predicative universe levels is *not* an instance of his framework — by reducing the problem to the one of finding a finite basis for the solutions of two linear problems in a specific max-plus algebra. To the best of our knowledge, it is however not known if the space of solutions for the problems we consider always admits a finite basis, and so the proof of correctness of our algorithm has to rely on this conjecture. Finally, we expect that this algorithm should have a high complexity, so to address this we propose to pre-process problems using our first constraint-postponement-based algorithm, allowing to cheaply eliminate some equations from the problem before running the second algorithm.

17.1 Preliminaries on levels

Before presenting our results, we first start by reviewing some important properties about universe levels that will be useful in our proofs.

In order to be able to compare levels syntactically, it is useful to introduce a notion of canonical form. A level is said to be in *canonical form* [Voe14, Gen20] when it is of the form

$$p \sqcup n_1 + x_1 \sqcup \cdots \sqcup n_k + x_k$$

with $n_i \leq p$ for all $i = 1, \ldots, k$, and each variable occurs only once. In this case we call p the *constant coefficient*, and n_i the *coefficient of* x_i. We recall the following fundamental property, which appears in [Voe14, Gen20, Bla22], and which we reprove here for completeness reasons.

Theorem 17.3. *Every level is equivalent to a canonical form, which is unique modulo associativity-commutativity. Moreover, there is a computable function mapping each level to one of its canonical forms.*

Proof. Let us describe the calculation of a canonical form, while at the same time showing that each step leads to a convertible level. Given a level t, we first replace each variable x by $x \sqcup 0$ (which are convertible levels). Then, by repeatedly applying $1+(t \sqcup u) \simeq 1+t \sqcup 1+u$, we get a level of the form $t_1 \sqcup \cdots \sqcup t_k$, in which each t_i is either of the form $n_i + x_i$ or n_i. Note that we can easily show $n+x \sqcup x \simeq n+x$ for all $n \in \mathbb{N}$, by induction on n, which then implies $n + x \sqcup m + x \simeq \max\{n, m\} + x$. Using this equation, we can merge all constant coefficients, and then all coefficients of a same variable, by always taking the maximum between them. Because in the beginning we started by replacing each variable x by $x \sqcup 0$, it follows that the constant coefficient of the resulting level must be greater or equal to all variable coefficients. Therefore, we have reached a canonical form.

To see that the canonical form is unique modulo associativity-commutativity, it suffices to note that if two canonical forms have different coefficients for a variable x, then by applying a substitution mapping x to some n large enough and the other variables to 0 we get two levels which are not convertible, hence the canonical forms we started with could not have been convertible. Similarly, if the constant coefficients are different, it suffices to take the substitution mapping all variables to 0, which then also yields non-convertible levels. ■

We hence get the following theorem, also in [Voe14, Gen20, Bla22].

Corollary 17.1. *The equational theory of predicative universe levels is decidable.*

In view of Theorem 17.3 and the notion of canonical form, we introduce the following notation: given a level t, we write $t\langle x\rangle$ for the coefficient of x in its canonical form, and set it to $-\infty$ if $x \notin \mathsf{fv}(t)$. We extend this notation to $t\langle\bullet\rangle$, which denotes the constant coefficient of the canonical form of t.

Remark 17.3. Note that, from the definition of canonical forms, we always have:

(i) $t\langle\bullet\rangle \neq -\infty$

(ii) $t\langle\bullet\rangle \geq t\langle x\rangle$ for all x

(iii) $t\langle x\rangle \neq -\infty$ only for finitely many x

Moreover, a function $\underline{\mathcal{V}} \cup \{\bullet\} \longrightarrow \mathbb{N} \cup \{-\infty\}$ defines a valid canonical form exactly when these three conditions are met. □

In the following, let $x_\bullet$ stand for either a variable x or the token $\bullet$. Then Theorem 17.3 says exactly that $t \simeq u$ iff for all $x_\bullet$ we have $t\langle x_\bullet\rangle = u\langle x_\bullet\rangle$. This principle, together with the following two easy lemmas, will be very useful when proving or disproving that two levels are equivalent.

Lemma 17.1. *The following equations hold for all t, u and $x_\bullet$.*

$$\begin{aligned}(t \sqcup u)\langle x_\bullet\rangle &= \max\{t\langle x_\bullet\rangle, u\langle x_\bullet\rangle\}\\ (n + u)\langle x_\bullet\rangle &= n + u\langle x_\bullet\rangle\end{aligned}$$

Proof. By merging the canonical forms of t and u like in the proof of Theorem 17.3 we get a canonical form (which by unicity must be *the* canonical form, modulo associativity-commutativity) with the coefficients given by the first equation. Similarly, by moving the $\sqcup$ outside in $n + u$ we also get a canonical form, with the coefficients described by the second equation. ■

Lemma 17.2. *The following equations hold for all t, θ, x and $Y \supseteq \mathsf{fv}(t)$.*[5]

$$\begin{aligned}t[\theta]\langle x\rangle &= \max\{t\langle y\rangle + y[\theta]\langle x\rangle \mid y \in Y\}\\ t[\theta]\langle \bullet\rangle &= \max(\{t\langle \bullet\rangle\} \cup \{t\langle y\rangle + y[\theta]\langle \bullet\rangle \mid y \in Y\})\end{aligned}$$

Proof. We start from t and θ in canonical form, we apply θ to t and then we rearange terms like in the proof of Theorem 17.3 by moving the $\sqcup$ outside and merging occurrences of the same variable, resulting in a canonical form. The coefficients we get the end are then precisely the ones described by the above equations. ■

Finally, the definition of $\simeq$ can now be justified by the following property. Given a function ϕ mapping each x to a natural number, define the interpretation $[\![t]\!]_\phi \in \mathbb{N}$ of a level t by interpreting the symbols $0, \mathsf{S}$ and $\sqcup$ as zero, successor and max, and by interpreting each variable x by $\phi(x)$.

Proposition 17.1. *We have $t \simeq u$ iff, for all ϕ, $[\![t]\!]_\phi = [\![u]\!]_\phi$.*

Proof. Note that for each axiom $t \approx u$ of the equational theory we have $[\![t]\!]_\phi = [\![u]\!]_\phi$ for all ϕ, and thus the direction $\Rightarrow$ can be showed by an easy induction on $t \simeq u$.

For the other direction, let us take the canonical forms t' of t and u' of u. By the left to right implication, we have $[\![t]\!]_\phi = [\![t']\!]_\phi$ and $[\![u]\!]_\phi = [\![u']\!]_\phi$ for all ϕ, hence $[\![t']\!]_\phi = [\![u']\!]_\phi$ for all ϕ. By varying ϕ over suitable valuations we can show that t' and u' have the same constant coefficients, and that each variable appearing in one also appears in the other with the same coefficient. Therefore, t' and u' are equal modulo associativity-commutativity, and thus $t \simeq u$. ■

[5] Note that the variables $y \notin \mathsf{fv}(t)$ do not change the value of the right-hand sides, given that their coefficients are $-\infty$.

In other words, $\simeq$ allows one to simplify level expressions which are semantically the same — for instance, $1 + x \sqcup x \sqcup 0$ and $1 + x$. This also shows that our definition of $\simeq$, which is also used in AGDA [Tea], agrees with the one used in other works about universe levels [Gen20, Voe14, Fer21, Bla22].

17.2 Characterizing equations that admit a m.g.u.

With the preliminaries now set up, we can move to the first main contribution of this chapter: a characterization of the equations that admit a most general unifier, along with an explicit description of a m.g.u. in these cases.

We start by extending the notion of canonical forms to equations: we say that $t \overset{?}{=} u$ is in *canonical form* when the following conditions are satisfied:

(1) Both t and u are in canonical form.

(2) If $x \in \mathsf{fv}(t) \cap \mathsf{fv}(u)$, then $t\langle x\rangle = u\langle x\rangle$

(3) There is some $x_\bullet \in \mathsf{fv}(t, u) \cup \{\bullet\}$ with $t\langle x_\bullet\rangle = 0$ or $u\langle x_\bullet\rangle = 0$

The main motivation for introducing this notion is the following result, stating that in our analysis it suffices to consider only equations in canonical form.

Proposition 17.2. *For all equations $t_1 \overset{?}{=} t_2$, there is an equation $u_1 \overset{?}{=} u_2$ in canonical form, such that for all θ, $t_1[\theta] \simeq t_2[\theta]$ iff $u_1[\theta] \simeq u_2[\theta]$.*

Proof. Let $t_1 \overset{?}{=} t_2$ be any equation. We apply transformations so that properties (1), (2) and (3) that define canonical forms are satisfied one by one, and we argue that they do not change the set of unifiers.

1. We put each level t_p in canonical form t'_p. It is clear that this preserves the set of unifiers, as any level is convertible to its canonical form.

2. If some variable x appears in t'_1 and t'_2 with different coefficients, we remove it from the side with smaller coefficient, and we name the resulting equation $t''_1 \overset{?}{=} t''_2$ — this step is then repeated until condition (2) of the canonical form definition is met. By decomposing $t'_1 \simeq v_1 \sqcup n + x$ and $t'_2 \simeq v_2 \sqcup m + x$ with $n < m$ (or the symmetric), the correctness of this step follows from $t'_1[\theta] \simeq t'_2[\theta]$ iff $v'_1[\theta] \sqcup n + x[\theta] \simeq v_2[\theta] \sqcup m + x[\theta]$ iff $v_1[\theta] \simeq v_2[\theta] \sqcup m + x[\theta]$, where the last equivalence follows from the fact that

$$\mathsf{max}\{k_1, n + q\} = \mathsf{max}\{k_2, m + q\} \qquad \Longleftrightarrow \qquad k_1 = \mathsf{max}\{k_2, m + q\}$$

for all $k_1, k_2, n, m, q \in \mathbb{N}$ with $n < m$, and then by Proposition 17.1.

3. Finally, if no coefficient in t_1'' or t_2'' is equal to zero, we subtract from all coefficients the value of the current minimal coefficient, and we name the resulting equation $t_1''' \overset{?}{=} t_2'''$. If we call this value k, then the correctness of this step follows from the fact that $t_p'' \simeq k + t_p'''$ for $p = 1, 2$, and so $t_1''[\theta] \simeq t_2''[\theta]$ iff $k + t_1'''[\theta] \simeq k + t_2'''[\theta]$ iff $t_1'''[\theta] \simeq t_2'''[\theta]$, where the last equivalence follows by applying Proposition 17.1.

It is clear that $t_1''' \overset{?}{=} t_2'''$ is in canonical form, and we have shown that each step of the transformation preserves the set of unifiers. ■

Example 17.4. Consider the equation $x \sqcup 1 + (x \sqcup 1 + y) \overset{?}{=} y \sqcup 2 + x$ and let us show how it can be put in canonical form using the underlying algorithm of the above proof. First, we compute the level canonical forms of each side, yielding

$$2 \sqcup 1 + x \sqcup 2 + y \overset{?}{=} 2 \sqcup 2 + x \sqcup y$$

As the variables x and y appear in the two sides with different coefficients, we then remove from each of the sides the occurrence with the smaller one, yielding

$$2 \sqcup 2 + y \overset{?}{=} 2 \sqcup 2 + x$$

Finally, as the minimum among all coefficients is 2, we subtract this from all of them, giving

$$0 \sqcup y \overset{?}{=} 0 \sqcup x$$

□

We are now able to state the main theorem that we are going to show. In the following, if $k \in \mathbb{N}$ we write $[k]$ for the set $\{1, ..., k\}$, and we call an equation $t_1 \overset{?}{=} t_2$ trivial when $t_1 \simeq t_2$. We also call $t_2 \overset{?}{=} t_1$ the *symmetric* of the equation $t_1 \overset{?}{=} t_2$. Finally, if $\vec{x} = x_1 ... x_k$ is a list of level variables, we sometimes identify it with the level $x_1 \sqcup ... \sqcup x_k$.

Theorem 17.5. *A non-trivial equation has*

(A) *a most general unifier iff its canonical form (or its symmetric) is of the form*

  (i) $n \sqcup x \overset{?}{=} t$ *with* $n < t\langle\bullet\rangle$, *in which case* $\theta := x \mapsto t$ *is a m.g.u.*

  (ii) $0 \sqcup \vec{x} \sqcup \vec{y} \overset{?}{=} 0 \sqcup \vec{x} \sqcup \vec{z}$ *with* $\vec{x}$, $\vec{y}$ *and* $\vec{z}$ *disjoint, in which case a m.g.u. is given by*

$$\begin{array}{rlr} \theta := & x_k \mapsto \alpha_k \sqcup (\sqcup_{n\in[p_1]}\beta_{k,n}) \sqcup (\sqcup_{m\in[p_2]}\gamma_{k,m}) & (k \in [p_0]) \\ & y_n \mapsto (\sqcup_{k\in[p_0]}\beta_{k,n}) \sqcup (\sqcup_{m\in[p_2]}\delta_{n,m}) & (n \in [p_1]) \\ & z_m \mapsto (\sqcup_{k\in[p_0]}\gamma_{k,m}) \sqcup (\sqcup_{n\in[p_1]}\delta_{n,m}) & (m \in [p_2]) \end{array}$$

where p_0, p_1, p_2 *are the lengths of* $\vec{x}$, $\vec{y}$ *and* $\vec{z}$ *respectively, and where* $\{\alpha_k\}_{k\in[p_0]}$, $\{\beta_{k,n}\}_{k\in[p_0],n\in[p_1]}$, $\{\gamma_{k,m}\}_{k\in[p_0],m\in[p_2]}$ *and* $\{\delta_{n,m}\}_{n\in[p_1],m\in[p_2]}$ *are disjoint sets of variables.*

(B) *no unifier iff its canonical form (or its symmetric) is of the form* $n \overset{?}{=} t$ *with* $n < t\langle\bullet\rangle$

(C) *some unifier but no most general one iff its canonical form (or its symmetric) is not of any of the previous forms*

Before proving the result, let us consider some examples to see how it can be used.

Example 17.6.

- The equation $x \sqcup y \overset{?}{=} x \sqcup z$ has the canonical form

  $$0 \sqcup x \sqcup y \overset{?}{=} 0 \sqcup x \sqcup z$$

  and therefore, by point (A).(ii) it admits the m.g.u.

  $$\theta = \{x \mapsto \alpha \sqcup \beta \sqcup \gamma,\ y \mapsto \beta \sqcup \delta,\ z \mapsto \gamma \sqcup \delta\}$$

- The equation $x \sqcup 1 + (y \sqcup 2) \overset{?}{=} 1 + (2 \sqcup x \sqcup y)$ has the canonical form

  $$2 \sqcup y \overset{?}{=} 2 \sqcup x \sqcup y$$

  and therefore by point (C) it is solvable but admits no m.g.u.

- The equation $x \sqcup 1 + (y \sqcup 1) \overset{?}{=} 1 + (2 \sqcup x \sqcup y)$ has the canonical form

  $$1 \sqcup y \overset{?}{=} 2 \sqcup x \sqcup y$$

  and therefore by point (A).(i) it admits the m.g.u.

  $$\theta = \{y \mapsto 2 \sqcup x \sqcup y\}$$

- The equation $x \sqcup 1 + (y \sqcup 1) \overset{?}{=} 2 + (1 \sqcup x \sqcup y)$ has the canonical form

  $$0 \overset{?}{=} 1 \sqcup x \sqcup y$$

  and therefore, by point (B) it admits no unifier. □

Let us now move to the proof of Theorem 17.5. Figure 17.1 shows its structure: we take a non-trivial equation in canonical form and consider its possible forms. Each leaf is annotated with the proposition associated with its proof, along with the case of Theorem 17.5 which we are in. We also write $|t_2|$ for the number of free variables occurring in t_2.

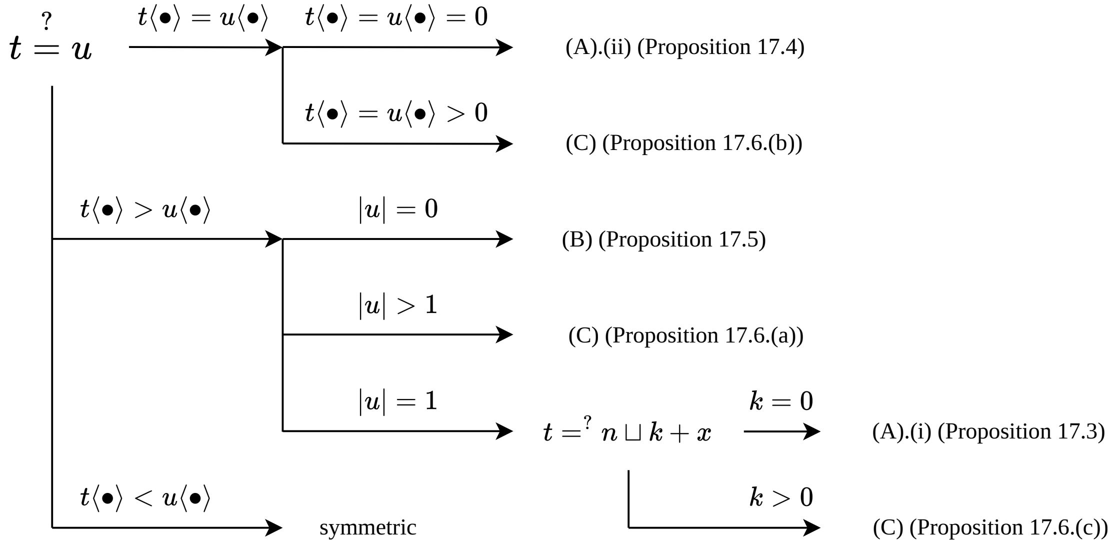

$t \overset{?}{=} u$
$t\langle\bullet\rangle = u\langle\bullet\rangle$
$t\langle\bullet\rangle = u\langle\bullet\rangle = 0$
(A).(ii) (Proposition 17.4)
$t\langle\bullet\rangle = u\langle\bullet\rangle > 0$
(C) (Proposition 17.6.(b))
$t\langle\bullet\rangle > u\langle\bullet\rangle$
$|u| = 0$
(B) (Proposition 17.5)
$|u| > 1$
(C) (Proposition 17.6.(a))
$|u| = 1$
$t =^? n \sqcup k + x$
$k = 0$
(A).(i) (Proposition 17.3)
$t\langle\bullet\rangle < u\langle\bullet\rangle$
symmetric
$k > 0$
(C) (Proposition 17.6.(c))

Figure 17.1: Structure of proof of Theorem 17.5

Equations with m.g.u.s

Proposition 17.3. *The equation in canonical form* $n \sqcup x \overset{?}{=} t$ *with* $n < t\langle\bullet\rangle$ *has the mgu* $\tau = \{x \mapsto t\}$.

Proof. It is easy to verify that τ is a unifier. Now let θ be any unifier and let us first show that $x[\theta] \simeq t[\theta]$. To do this we show $x[\theta]\langle y_\bullet\rangle = t[\theta]\langle y_\bullet\rangle$ for all $y_\bullet$. Using Lemma 17.1, we have $t[\theta]\langle y_\bullet\rangle = (n \sqcup x[\theta])\langle y_\bullet\rangle = \max\{n\langle y_\bullet\rangle, x[\theta]\langle y_\bullet\rangle\}$ for all $y_\bullet$, and because $n\langle y\rangle = 0$, then it follows that $x[\theta]\langle y\rangle = t[\theta]\langle y\rangle$ for all y. Then, for the case $y_\bullet = \bullet$, we have $\max\{n, x[\theta]\langle\bullet\rangle\} = t[\theta]\langle\bullet\rangle$, but because $t[\theta]\langle\bullet\rangle > n$ we must have $x[\theta]\langle\bullet\rangle = t[\theta]\langle\bullet\rangle$. Now we can show that τ is more general than θ: we have $y[\tau][\theta] \simeq y[\theta]$ trivially for $y \neq x$, and for $y = x$ we have $x[\tau][\theta] = t[\theta] \simeq x[\theta]$. ∎

Proposition 17.4. *The equation* $0 \sqcup \vec{x} \sqcup \vec{y} \overset{?}{=} 0 \sqcup \vec{x} \sqcup \vec{z}$ *with* $\vec{x}$, $\vec{y}$ *and* $\vec{z}$ *disjoint has the m.g.u.*

$$\begin{aligned} \theta := \quad & x_k \mapsto \alpha_k \sqcup (\sqcup_{n\in[p_1]}\beta_{k,n}) \sqcup (\sqcup_{m\in[p_2]}\gamma_{k,m}) && (k \in [p_0]) \\ & y_n \mapsto (\sqcup_{k\in[p_0]}\beta_{k,n}) \sqcup (\sqcup_{m\in[p_2]}\delta_{n,m}) && (n \in [p_1]) \\ & z_m \mapsto (\sqcup_{k\in[p_0]}\gamma_{k,m}) \sqcup (\sqcup_{n\in[p_1]}\delta_{n,m}) && (m \in [p_2]) \end{aligned}$$

where p_0, p_1, p_2 *are the lengths of* $\vec{x}, \vec{y}$ *and* $\vec{z}$ *respectively, and where* $\{\alpha_k\}_{k\in[p_0]}$, $\{\beta_{k,n}\}_{k\in[p_0],n\in[p_1]}$, $\{\gamma_{k,m}\}_{k\in[p_0],m\in[p_2]}$ *and* $\{\delta_{n,m}\}_{n\in[p_1],m\in[p_2]}$ *are disjoint sets of variables.*

Proof. It is easy to see that θ is a unifier: all introduced variables appear in both sides, with coefficient 0. Given a unifier τ, define τ' by setting for each $x_\bullet$:

$$\begin{aligned}
\alpha_k[\tau']\langle x_\bullet\rangle &:= x_k[\tau]\langle x_\bullet\rangle \\
\beta_{k,n}[\tau']\langle x_\bullet\rangle &:= \min\{x_k[\tau]\langle x_\bullet\rangle, y_n[\tau]\langle x_\bullet\rangle\} \\
\gamma_{k,m}[\tau']\langle x_\bullet\rangle &:= \min\{x_k[\tau]\langle x_\bullet\rangle, z_m[\tau]\langle x_\bullet\rangle\} \\
\delta_{n,m}[\tau']\langle x_\bullet\rangle &:= \min\{y_n[\tau]\langle x_\bullet\rangle, z_m[\tau]\langle x_\bullet\rangle\}
\end{aligned}$$

The reader can verify that conditions (i), (ii) and (iii) of Remark 17.3 are satisfied, so this definition is indeed valid. Let us now show that τ and $\tau' \circ \theta$ are convertible over the variables $\vec{x}, \vec{y}, \vec{z}$.

Because $x_k[\tau]\langle x_\bullet\rangle$ is greater or equal than $\min\{x_k[\tau]\langle x_\bullet\rangle, y_n[\tau]\langle x_\bullet\rangle\}$ and $\min\{x_k[\tau]\langle x_\bullet\rangle, z_m[\tau]\langle x_\bullet\rangle\}$ for all n, m, using Lemma 17.1 we have

$$\begin{aligned}
x_k[\theta][\tau']\langle x_\bullet\rangle &= (\alpha_k[\tau'] \sqcup (\sqcup_{n\in[p_1]}\beta_{k,n}[\tau']) \sqcup (\sqcup_{m\in[p_2]}\gamma_{k,m}[\tau']))\langle x_\bullet\rangle \\
&= \max(\{x_k[\tau]\langle x_\bullet\rangle\} \\
&\qquad \cup \{\min\{x_k[\tau]\langle x_\bullet\rangle, y_n[\tau]\langle x_\bullet\rangle\} \mid n \in [p_1]\} \\
&\qquad \cup \{\min\{x_k[\tau]\langle x_\bullet\rangle, z_m[\tau]\langle x_\bullet\rangle\} \mid m \in [p_2]\}) \\
&= x_k[\tau]\langle x_\bullet\rangle
\end{aligned}$$

for all $x_\bullet$ and $k \in [p_0]$. Therefore, we get $x_k[\theta][\tau'] \simeq x_k[\tau]$ for all $k \in [p_0]$.

Because τ is a unifier, we have $0 \sqcup \vec{x}[\tau] \sqcup \vec{y}[\tau] \simeq 0 \sqcup \vec{x}[\tau] \sqcup \vec{z}[\tau]$, which together with Lemma 17.1 yields

$$\begin{aligned}
&\max(\{x_k[\tau]\langle x_\bullet\rangle \mid k \in [p_0]\} \cup \{y_n[\tau]\langle x_\bullet\rangle \mid n \in [p_1]\}) \\
= \;&\max(\{x_k[\tau]\langle x_\bullet\rangle \mid k \in [p_0]\} \cup \{z_m[\tau]\langle x_\bullet\rangle \mid m \in [p_2]\})
\end{aligned}$$

for all $x_\bullet$. Therefore, for every $n \in [p_1]$, there is some $k \in [p_0]$ with $x_k[\tau]\langle x_\bullet\rangle \geq y_n[\tau]\langle x_\bullet\rangle$, or there is some $m \in [p_2]$ with $z_m[\tau]\langle x_\bullet\rangle \geq y_n[\tau]\langle x_\bullet\rangle$. Hence, either we have $\min\{x_k[\tau]\langle x_\bullet\rangle, y_n[\tau]\langle x_\bullet\rangle\} = y_n[\tau]\langle x_\bullet\rangle$ for some $k \in [p_1]$, or we have $\min\{y_n[\tau]\langle x_\bullet\rangle, z_m[\tau]\langle x_\bullet\rangle\} = y_n[\tau]\langle x_\bullet\rangle$ for some $m \in [p_2]$. Therefore, using Lemma 17.1, we obtain

$$\begin{aligned}
y_n[\theta][\tau']\langle x_\bullet\rangle &= ((\sqcup_{k\in[p_0]}\beta_{k,n}[\tau']) \sqcup (\sqcup_{m\in[p_2]}\delta_{n,m}[\tau']))\langle x_\bullet\rangle \\
&= \max(\{\min\{x_k[\tau]\langle x_\bullet\rangle, y_n[\tau]\langle x_\bullet\rangle\} \mid k \in [p_0]\} \\
&\qquad \cup \{\min\{y_n[\tau]\langle x_\bullet\rangle, z_m[\tau]\langle x_\bullet\rangle\} \mid m \in [p_2]\}) \\
&= y_n[\tau]\langle x_\bullet\rangle
\end{aligned}$$

for all $x_\bullet$ and $n \in [p_1]$. Therefore, we get $y_n[\theta][\tau'] \simeq y_n[\tau]$ for all $n \in [p_1]$.

Finally, a symmetrical reasoning shows $z_m[\theta][\tau'] \simeq z_m[\tau]$ for all $m \in [p_2]$. ■

Remark 17.4. Proposition 17.4 shows that, when there is no occurrence of S in the equation, it can be solved as an ACUI unification problem. Indeed, the m.g.u. given there is also a m.g.u. of the equation when seen as a unification problem in the theory ACUI [BB88]. □

Unsolvable equations

Proposition 17.5. *A non-trivial equation in canonical form has no solution iff it (or its symmetric) is of the form $m \overset{?}{=} t$ with $m < t\langle\bullet\rangle$.*

Proof. It is clear that $m \overset{?}{=} t$ with $m < t\langle\bullet\rangle$ has no solution. For the other direction, we show that any equation not of this form has a solution.

First note that if $t_1 \overset{?}{=} u_2$ has variables in both sides then it is easy to build a solution. Indeed, if $x_1 \in \mathsf{fv}(t_1)$, $x_2 \in \mathsf{fv}(t_2)$ are (not necessarily distinct) variables, then $\theta = x_1 \mapsto p - t_1\langle x_1\rangle,\ x_2 \mapsto p - t_2\langle x_2\rangle,\ \_ \mapsto 0$,[6] where $p = \max\{t_1\langle\bullet\rangle, t_2\langle\bullet\rangle\}$, is a solution — note that this is also well-defined in the case $x_1 = x_2$, because for equations in canonical form this implies $t_1\langle x_1\rangle = t_2\langle x_2\rangle$.

We can thus restrict our analysis to equations with one of the sides constant, of the form $m \overset{?}{=} t$. Note that we can suppose that t has some variable: indeed, if t is constant and equal to m then the equation is trivial, and if t is constant and different from m then the equation is of the form $m' \overset{?}{=} t'$ with $m' < t'\langle\bullet\rangle$ and indeed has no solution. Finally, it is easy to see that for $m \overset{?}{=} t$ with $m \geq t\langle\bullet\rangle$ and where t has some variable x, we have the unifier $\theta = x \mapsto m - t\langle x\rangle,\ \_ \mapsto 0$. ■

Solvable equations not admitting a m.g.u.

The last ingredient for our proof is showing that in all other cases there is no m.g.u. In order to show this, we will use the following auxiliary lemma. In the following, we refer to a level not containing any occurrence of S as *flat*.

Lemma 17.3 (Auxiliary lemma). *Let $t \overset{?}{=} u$ be an equation admitting a unifer θ and a m.g.u. τ.*

(1) *If $x[\theta] = 0$ and $x \in \mathsf{fv}(t) \cup \mathsf{fv}(u)$ then $x[\tau]$ is flat.*

(2) *If $x[\theta] = m > 0$ with $x \in \mathsf{fv}(t) \cup \mathsf{fv}(u)$, and for all $i = 1, \ldots, k$ we have $x_i[\theta] = n_i < m$ and $x_i \in \mathsf{fv}(t) \cup \mathsf{fv}(u)$, then if $x[\tau]$ is flat it must contains one variable not in any $x_i[\tau]$.*

(3) *If $x \in \mathsf{fv}(t)$ and $z \in \mathsf{fv}(x[\theta])$, then for some $y \in \mathsf{fv}(u)$ we must have $z \in \mathsf{fv}(y[\theta])$.*

Proof. We show each point separately.

1. Because τ is a m.g.u., for some θ' we have $x[\tau][\theta'] \simeq x[\theta] = 0$, so by Lemma 17.2 we conclude that all coefficients of $x[\tau]$ must be either 0 or $-\infty$.

[6] Recall that substitutions are *finite* sets of pairs of the form $x \mapsto t$, so $\_ \mapsto \ldots$ is actually an abuse of notation, which should be understood as "every other relevant variable is mapped to . . . ".

2. Because τ is a m.g.u., for some θ' we have $x[\tau][\theta'] \simeq x[\theta] = m$ and $x_i[\tau][\theta'] \simeq x_i[\theta] = n_i < m$ for $i = 1, \dots, k$. Now if we suppose that $x[\tau]$ is flat, by Lemma 17.2 the only way to have $x[\tau][\theta'] \simeq m > 0$ is if for some variable y in $x[\tau]$ we have $y[\theta']\langle \bullet \rangle = m$. But because $x_i[\tau][\theta'] \simeq n_i < m$, it is clear that y cannot appear in any of the $x_i[\tau]$.

3. Follows from the fact that, if θ is a unifier, then the same variables that appear in $t_1[\theta]$ must also appear in $u_2[\theta]$. ■

Proposition 17.6 (Equations with no mgu). *The following non-trivial equations in canonical form do not admit a m.g.u.:*

(a) $t \overset{?}{=} u$ *with* $|u| > 1$ *and* $t\langle \bullet \rangle > u\langle \bullet \rangle$.

(b) $t \overset{?}{=} u$ *with* $t\langle \bullet \rangle = u\langle \bullet \rangle > 0$.

(c) $t \overset{?}{=} n \sqcup k + x$ *with* $k > 0$ *and* $n < t\langle \bullet \rangle$.

Proof. The structure of the proof is the same in all cases: we suppose the existence of a most general unifier τ which we use to obtain a contradiction.

(a) Let x, y be two different variables in u. By Lemma 17.3.(1), the unifiers $\theta_1 = x \mapsto t\langle \bullet \rangle - u\langle x \rangle,\ \_ \mapsto 0$ and $\theta_2 = y \mapsto t\langle \bullet \rangle - u\langle y \rangle,\ \_ \mapsto 0$ show that $z[\tau]$ is flat for all $z \in \mathrm{fv}(t, u)$. But then we have $t[\tau][\_ \mapsto 0] \simeq t\langle \bullet \rangle$ and $u[\tau][\_ \mapsto 0] \simeq u\langle \bullet \rangle$, and because τ is a unifier we must then have $t\langle \bullet \rangle = u\langle \bullet \rangle$, a contradiction with $t\langle \bullet \rangle > u\langle \bullet \rangle$.

(b) First note that $\_ \mapsto 0$ is a unifier, so by Lemma 17.3.(1), $z[\tau]$ is flat for all $z \in \mathrm{fv}(t, u)$. Because the equation is supposed to be in canonical form and non-trivial, some variable x appears in only one side. Take such a x with a minimal coefficient, which we henceforth call p.

If $p < t\langle \bullet \rangle = u\langle \bullet \rangle$, then the unifier $\theta_1 = x \mapsto t\langle \bullet \rangle - p,\ \_ \mapsto 0$ shows, by Lemma 17.3.(2), that $x[\tau]$ contains a variable not in any other $y[\tau]$ with $y \in \mathrm{fv}(t, u) \setminus \{x\}$, a contradiction with Lemma 17.3.(3), as x appears in only one side.

Suppose now that $p = t\langle \bullet \rangle = u\langle \bullet \rangle$. Because the equation is in canonical form and the constant coefficient of each side is different from 0, then some variable y must appear with coefficient 0. Moreover, because the minimal coefficient of a variable occurring in only one side is $p \neq 0$, it follows that y must appear in both sides (both occurrences, of course, with coefficient 0). Because $p = t\langle \bullet \rangle > 0$, by Lemma 17.3.(2) the unifier $\theta_2 = x \mapsto 1,\ y \mapsto p + 1,\ \_ \mapsto 0$ shows that some variable $x' \in \mathrm{fv}(x[\tau])$ does not appear in any $z[\tau]$ with $z \in \mathrm{fv}(t, u) \setminus \{x, y\}$. Therefore, because x' can only also occur in $y[\tau]$, and because the coefficient of x is p and the coefficient of y is 0, by composing τ with $x' \mapsto 1,\ \_ \mapsto 0$ we get $p + 1$ at the side in which x occurs but p at the other side, a contradiction.

(c) Because we suppose the equation is in canonical form, we must have some $y \in \mathsf{fv}(t)$ different from x with coefficient 0. By Lemma 17.3.(1), the unifier $\theta_1 = x \mapsto t\langle\bullet\rangle - k,\ \_ \mapsto 0$ shows that $y[\tau]$ is flat, and by Lemma 17.3.(2) the unifier $\theta_2 = x \mapsto t\langle\bullet\rangle - k,\ y \mapsto t\langle\bullet\rangle,\ \_ \mapsto 0$ shows that some variable in $y[\tau]$ does not occur in $x[\tau]$, given that $t\langle\bullet\rangle - k < t\langle\bullet\rangle$. Because x is the only variable that appears in the right side, this establishes a contradiction with Lemma 17.3.(3). ■

Putting everything together

Proof of Theorem 17.5. We proceed as illustrated in Figure 17.1. The case $t\langle\bullet\rangle = u\langle\bullet\rangle$ is covered by Proposition 17.4 when $t\langle\bullet\rangle = u\langle\bullet\rangle = 0$, and by Proposition 17.6.(b) when $t\langle\bullet\rangle = u\langle\bullet\rangle \neq 0$. In the case $t\langle\bullet\rangle \neq u\langle\bullet\rangle$ we suppose w.l.o.g. that $t\langle\bullet\rangle > u\langle\bullet\rangle$, the other case being symmetric. Then we branch on the number of variables occurring in u: the case of no variables is covered by Proposition 17.5, and the case of more than one variable is covered by Proposition 17.6.(a). For the case of exactly one variable, we branch on the coefficient of this one variable. If the coefficient is zero, the result follows from Proposition 17.3, otherwise it follows by Proposition 17.6.(c). ■

17.3 A partial unification algorithm

We now apply Theorem 17.5 in the design of a partial algorithm for universe level unification. The *configurations* of our algorithm are either of the form $\bot$, or $C; \theta$ with $\mathsf{dom}(\theta)$ and $\mathsf{fv}(C) \cup \mathsf{vrange}(\theta)$ disjoint, where $\mathsf{vrange}(\theta) := \cup_{x \in \mathsf{dom}(\theta)} \mathsf{fv}(x[\theta])$. Configurations are rewritten according to the rules of Figure 17.2, where $C[\sigma] := \{t_1[\sigma] \overset{?}{=} t_2[\sigma] \mid t_1 \overset{?}{=} t_2 \in C\}$ and $\theta[\sigma] := \{x \mapsto x[\theta][\sigma] \mid x \in \mathsf{dom}(\theta)\}$. We define the variables of a configuration $C; \theta$ by $\mathcal{V}(C; \theta) := \mathsf{fv}(C) \cup \mathsf{dom}(\theta)$.

Theorem 17.5 is used in Solve to detect if some equation admits a m.g.u., and in Fail to detect if some equation is unsolvable. We assume that for each σ chosen in step Solve we have $\mathsf{dom}(\sigma) = \mathsf{fv}(t_1, t_2)$, which guarantees that $\mathcal{V}(C; \theta) \subseteq \mathcal{V}(C'; \theta')$ whenever

$$\boxed{C; \theta \rightsquigarrow C'; \theta'}$$

$$(\textsc{Solve}) \qquad \{t_1 \overset{?}{=} t_2\} \cup C; \theta \rightsquigarrow C[\sigma]; \sigma \cup \theta[\sigma] \qquad \text{if } \sigma = \mathsf{mgu}(t_1, t_2)$$

$$(\textsc{Fail}) \qquad \{t_1 \overset{?}{=} t_2\} \cup C; \theta \rightsquigarrow \bot \qquad \text{if } t_1 \text{ and } t_2 \text{ are not unifiable}$$

Figure 17.2: Unification algorithm for universe levels

$C; \theta \rightsquigarrow C'; \theta'$.[7] This assumption can always be satisfied by removing useless entries $x \mapsto t$ in σ for which $x \notin \mathsf{fv}(t_1, t_2)$, and adding trivial entries $x \mapsto x'$ for $x \in \mathsf{fv}(t_1, t_2) \setminus \mathsf{dom}(\sigma)$ and x' fresh, and the resulting substitution is still a m.g.u. We also suppose that the set $\mathsf{vrange}(\sigma)$ only contains fresh variables not in $\mathsf{fv}(C) \cup \mathsf{dom}(\theta) \cup \mathsf{vrange}(\theta)$, which also guarantees that steps preserve disjointness of $\mathsf{dom}(\theta)$ and $\mathsf{fv}(C) \cup \mathsf{vrange}(\theta)$. This assumption can always be satisfied by composing σ with a bijective renaming, as the composition of a m.g.u. with a bijective renaming is also a m.g.u.

The algorithm succeeds if it reaches a configuration of the form $\emptyset; \theta$, it fails if it reaches the configuration $\bot$ and it gets stuck if it reaches any other configuration in which no rule applies. Moreover, the following straightforward result guarantees that the algorithm cannot run forever, so these are the only options.

Proposition 17.7. *The relation $\rightsquigarrow$ defined in Figure 17.2 is strongly normalizing.*

Proof. Each step Solve decreases the cardinality of C, and a step Fail leads to a final state. ■

In the following, we write $t_1 \overset{?}{=} t_2 \in C; \theta$ when either $t_1 \overset{?}{=} t_2 \in C$ or $t_1 = x$ and $t_2 = u$ for some $x \mapsto u \in \theta$. We then write $\tau \vDash C; \theta$ when $t_1[\tau] \simeq t_2[\tau]$ for every $t_1 \overset{?}{=} t_2 \in C; \theta$. Finally, given substitutions τ, τ' and a set of variables X, we write $\tau =_X \tau'$ if $x[\tau] = x[\tau']$ for all $x \in X$.

Lemma 17.4 (Key lemma). *Suppose $C_1; \theta_1 \rightsquigarrow C_2; \theta_2$. Then*

1. *$\tau \vDash C_1; \theta_1$ implies $\tau' \vDash C_2; \theta_2$ for some τ' with $\tau' =_{\mathcal{V}(C_1;\theta_1)} \tau$*
2. *$\tau \vDash C_2; \theta_2$ implies $\tau \vDash C_1; \theta_1$*

Proof. The only possible case is rule Solve:

$$\{t_1 \overset{?}{=} t_2\} \cup C; \theta \rightsquigarrow C[\sigma]; \sigma \cup \theta[\sigma]$$

where σ is a m.g.u. of t_1 and t_2. We show each point separately.

1. By hypothesis we have $t_1[\tau] \simeq t_2[\tau]$, so because σ is a m.g.u. for t_1 and t_2 it follows that for some θ' we have $x[\sigma][\theta'] \simeq x[\tau]$ for all $x \in \mathsf{fv}(t_1, t_2)$. In the following, we suppose w.l.o.g. that $\mathsf{dom}(\theta') = \mathsf{vrange}(\sigma)$ — otherwise we just take the restriction of θ' to $\mathsf{vrange}(\sigma)$ and add trivial entries $x \mapsto x$ for $x \in \mathsf{vrange}(\sigma) \setminus \mathsf{dom}(\theta')$, and the equation $x[\sigma][\theta'] \simeq x[\tau]$ still holds. Recall as well that, by hypothesis on how m.g.u.s are chosen in Solve, $\mathsf{vrange}(\sigma)$ contains only fresh variables not in $\{t_1 \overset{?}{=} t_2\} \cup C; \theta$, and $\mathsf{dom}(\sigma) = \mathsf{fv}(t_1, t_2)$.

[7] Note that this property is not true in Robinson's unification algorithm, because the step eliminating a trivial equation $\{x \overset{?}{=} x\} \cup C; \theta \rightsquigarrow C; \theta$ may decrease the set of free variables of the configuration.

Defining $\tau' := \theta' \cup \{x \mapsto x[\tau] \mid x \in \mathsf{dom}(\tau) \setminus \mathsf{dom}(\theta')\}$, we first claim that $x[\sigma][\tau'] \simeq x[\tau]$ for all $x \notin \mathsf{vrange}(\sigma)$. To see this, first consider the case $x \in \mathsf{dom}(\sigma)$: then $x[\sigma]$ contains only variables in $\mathsf{vrange}(\sigma) = \mathsf{dom}(\theta')$ for which τ' because just θ', and so $x[\sigma][\tau'] = x[\sigma][\theta'] \simeq x[\tau]$. Now consider the case $x \notin \mathsf{dom}(\sigma)$: then $x[\sigma][\tau'] = x[\tau']$, and because we have $x \notin \mathsf{vrange}(\sigma) = \mathsf{dom}(\theta')$ then $x[\tau'] = x[\tau]$, concluding the proof of the claim.

By the above claim and the fact that $\mathsf{vrange}(\sigma)$ contains only fresh variables not in $C; \theta$, for each $u_1 \overset{?}{=} u_2 \in C; \theta$ we have $u_i[\tau] \simeq u_i[\tau' \circ \sigma]$ for $i = 1, 2$, so $\tau \vDash C; \theta$ implies $\tau' \circ \sigma \vDash C; \theta$ and thus $\tau' \vDash C[\sigma]; \theta[\sigma]$. Finally, the claim also implies $x[\sigma][\tau'] \simeq x[\tau']$ for all $x \in \mathsf{dom}(\sigma)$, given that $x[\tau] = x[\tau']$ for x not fresh. Hence, we conclude $\tau' \vDash C[\sigma]; \sigma \cup \theta[\sigma]$ as required.

2. By hypothesis we have $x[\tau] \simeq x[\sigma][\tau]$ for all $x \in \mathsf{dom}(\sigma)$, and the equation trivially holds for $x \notin \mathsf{dom}(\sigma)$, given that in this case $x[\sigma] = x$. Therefore, from $\tau \vDash C[\sigma]; \theta[\sigma]$ we get $\tau \vDash C; \theta$. Finally, because σ unifies t_1 and t_2, we have $t_1[\sigma][\tau] \simeq t_2[\sigma][\tau]$, and because $t_i[\tau] \simeq t_i[\sigma][\tau]$ for $i = 1, 2$, we get $t_1[\tau] \simeq t_2[\tau]$. We thus conclude $\tau \vDash \{t_1 \overset{?}{=} t_2\} \cup C; \theta$. ■

The key lemma then leads to the correctness of the unification algorithm.

Theorem 17.7 (Correctness of unification). *If $C; \emptyset \rightsquigarrow^* \emptyset; \theta$ then θ is a most general unifer for C, and if $C; \emptyset \rightsquigarrow^* \bot$ then C has no unifier.*

Proof. Suppose that $C; \emptyset \rightsquigarrow^* \emptyset; \theta$. Because $\mathsf{dom}(\theta)$ is disjoint from $\mathsf{vrange}(\theta)$, it follows that θ is idempotent, and thus $\theta \vDash \emptyset; \theta$. By iterating Lemma 17.4 we then get $\theta \vDash C; \emptyset$, showing that θ is a unifier for C. To see it is a most general one, given some unifier τ, we first iterate Lemma 17.4 to get $\tau' \vDash \emptyset; \theta$ for some $\tau' =_{\mathsf{fv}(C)} \tau$, implying $x[\tau'] \simeq x[\theta][\tau']$ for all $x \in \mathsf{dom}(\theta)$. But we have $\mathsf{fv}(C) = \mathcal{V}(C; \emptyset) \subseteq \mathcal{V}(\emptyset; \theta) = \mathsf{dom}(\theta)$, and moreover τ and τ' agree on $\mathsf{fv}(C)$, so we conclude $x[\tau] \simeq x[\theta][\tau']$ for all $x \in \mathsf{fv}(C)$.

Now suppose that $C; \emptyset \rightsquigarrow^* \bot$. Then we have $C; \emptyset \rightsquigarrow^* C'; \theta' \rightsquigarrow \bot$. If τ is a unifier for C, then by iterating Lemma 17.4 with the restriction of τ to $\mathsf{fv}(C)$, we get a unifier for C'. But if $C'; \theta' \rightsquigarrow \bot$, then C' must contain an unsolvable equation, a contradiction. ■

Remark 17.5. We note that the correctness proofs do not rely on any specificity of the equational theory of universe levels, and therefore the algorithm of Figure 17.2 can be used with any equational theory in which one can compute a m.g.u. for two terms when it exists. □

Because our algorithm uses Theorem 17.5, which gives a complete characterization of the equations that admit a m.g.u., it follows that our algorithm is complete for solving equations, in the sense that it can always find a m.g.u. for an equation that admits one. We can then wonder whether if it is also complete for problems that contain more than one equation. The following example shows that this is not the case.

Example 17.1. Consider the problem $C := \{1 + x \stackrel{?}{=} z \sqcup 1 + y,\ 1 + x \stackrel{?}{=} y \sqcup 1 + z\}$. We can check that, according to Theorem 17.5, both equations are solvable but admit no most general unifiers, so neither the rule Solve nor Fail apply. Nevertheless, by combining both equations we get $z \sqcup 1 + y \stackrel{?}{=} y \sqcup 1 + z$, whose canonical form is $0 \sqcup y \stackrel{?}{=} 0 \sqcup z$. Therefore, C is equivalent to $C \cup \{0 \sqcup y \stackrel{?}{=} 0 \sqcup z\}$, a problem that can be solved by our algorithm, yielding the m.g.u. $\theta = y \mapsto 0 \sqcup z,\ x \mapsto 0 \sqcup z$. It follows that C also admits θ as a m.g.u., yet our algorithm does not return any m.g.u., showing it is not complete for problems with more than one equation. □

Moreover, Theorem 17.2 shows that, even if our algorithm were complete, it would still get stuck in problems which are solvable but admit no m.g.u. In practice, it is very unsatisfying for the unification to get stuck, as this means that the whole predicativization algorithm has to halt. Thus, in order to prevent this, in our implementation we extended the unification with heuristics that are *only* applied when none of the presented rules applies. Then, whenever the heuristics are applied, the computed substitution is still a unifier, but might not be a most general one. This means that the term which generated the unification problem can still be translated to a valid term in $\mathbb{T}^{\forall}_{\mathbf{P}}$, but the resulting term might not be a most general universe-polymorphic instance.

17.4 Towards a complete unification algorithm

In this final section, we present a unification algorithm which we prove to be complete under two conjectures about max-plus algebra. Our algorithm works by reducing a unification problem to two problems in the semi-ring $\overline{\mathbb{N}}_{+} := (\overline{\mathbb{N}}, \oplus, \otimes)$, where $\overline{\mathbb{N}} := \mathbb{N} \cup \{-\infty\}$, $x \oplus y := \max\{x, y\}$ and $x \otimes y := x + y$. A finite basis for the solutions of the problems then allows us to provide a complete set of unifiers for the unification problem we started with.

More precisely, given an unification problem C with n equations — which we write as $t_1 \stackrel{?}{=} u_1, \ldots, t_i \stackrel{?}{=} u_i, \ldots, t_n \stackrel{?}{=} u_n$ — and m variables — which we write as $x_1, \ldots, x_j, \ldots, x_m$ — let us start by defining the matrices $\mathbf{T}, \mathbf{U} \in \overline{\mathbb{N}}_{n\times m}$ and $\mathbf{t}, \mathbf{u} \in \mathbb{N}_n$ by

$$\mathbf{t}_i := t_i\langle\bullet\rangle \qquad \mathbf{u}_i := u_i\langle\bullet\rangle \qquad \mathbf{T}_{i,j} := t_i\langle x_j\rangle \qquad \mathbf{U}_{i,j} := u_i\langle x_j\rangle$$

We then consider the following *two-sided linear problems* Φ_C and Ψ_C, in which $\mathbf{x} \in \overline{\mathbb{N}}_m$ is the unknown.

$$\mathbf{t} \oplus \mathbf{T} \otimes \mathbf{x} = \mathbf{u} \oplus \mathbf{U} \otimes \mathbf{x} \tag{Φ_C}$$

$$\mathbf{T} \otimes \mathbf{x} = \mathbf{U} \otimes \mathbf{x} \tag{Ψ_C}$$

Example 17.2. In this section we use as a running example the unification problem given by the only equation $1 \sqcup x_1 \stackrel{?}{=} 2 \sqcup 1 + x_2 \sqcup x_3$. This problem is translated into the matrices $\mathbf{T} = [0\ \ -\infty\ \ -\infty]$, $\mathbf{U} = [-\infty\ 1\ 0]$, $\mathbf{t} = [1]$ and $\mathbf{u} = [2]$. □

Our interest in these two problems is justified by the following result, where we write $\theta\langle y\rangle \in \overline{\mathbb{N}}_m$ for $(x_1[\theta]\langle y\rangle, \ldots, x_m[\theta]\langle y\rangle)$ and $\theta\langle\bullet\rangle \in \mathbb{N}_m$ for $(x_1[\theta]\langle\bullet\rangle, \ldots, x_m[\theta]\langle\bullet\rangle)$.

Proposition 17.8. *A substitution θ is a unifier for C iff $\theta\langle\bullet\rangle$ is a solution for Φ_C and $\theta\langle y\rangle$ is a solution for Ψ_C for all $y \in \mathsf{fv}(C[\theta])$.*

Proof. A substitution θ is a unifier iff, for all $t_i \overset{?}{=} u_i \in C$, we have $t_i[\theta]\langle\bullet\rangle = u_i[\theta]\langle\bullet\rangle$ and $t_i[\theta]\langle y\rangle = u_i[\theta]\langle y\rangle$ for all $y \in \mathsf{fv}(C[\theta])$. By Lemma 17.2, this holds iff we have

$$t_i\langle\bullet\rangle \oplus (\bigoplus_j t_i\langle x_j\rangle \otimes x_j[\theta]\langle\bullet\rangle) = u_i\langle\bullet\rangle \oplus (\bigoplus_j u_i\langle x_j\rangle \otimes x_j[\theta]\langle\bullet\rangle)$$

and

$$\bigoplus_j t_i\langle x_j\rangle \otimes x_j[\theta]\langle y\rangle = \bigoplus_j u_i\langle x_j\rangle \otimes x_j[\theta]\langle y\rangle$$

which are precisely the result of replacing $\theta\langle\bullet\rangle$ for $\mathbf{x}$ in Ψ_C and $\theta\langle y\rangle$ for $\mathbf{x}$ in Φ_C. ■

As we will see, in order to get a complete set of unifiers for C, we will need to compute finite bases for the space of solutions for the problems Ψ_C and Φ_C. This is the point in which we will need to introduce the following conjectures.[8]

Conjecture 17.1. *For all $\mathbf{A}, \mathbf{B} \in \overline{\mathbb{N}}_{n\times m}$, the set of solutions for*

$$\mathbf{A} \otimes \mathbf{x} = \mathbf{B} \otimes \mathbf{x}$$

is of the form $\{\bigoplus_{\mathbf{w}\in\mathcal{W}} r_{\mathbf{w}} \otimes \mathbf{w} \mid r_{\mathbf{w}} \in \overline{\mathbb{N}}$ for all $\mathbf{w} \in \mathcal{W}\}$ for some finite and computable set $\mathcal{W} \subset \overline{\mathbb{N}}_m$.

Conjecture 17.2. *For all $\mathbf{A}, \mathbf{B} \in \overline{\mathbb{N}}_{n\times m}$ and $\mathbf{p}, \mathbf{q} \in \overline{\mathbb{N}}_n$, the set of solutions for*

$$\mathbf{A} \otimes \mathbf{x} \oplus \mathbf{p} = \mathbf{B} \otimes \mathbf{x} \oplus \mathbf{q}$$

is of the form $\{\mathbf{a} \oplus \mathbf{v} \mid \mathbf{a} \in \mathcal{A}, \mathbf{v} \in X\}$ for some finite and computable set $\mathcal{A} \subset \overline{\mathbb{N}}_m$, and where X is the set of solutions for $\mathbf{A} \otimes \mathbf{x} = \mathbf{B} \otimes \mathbf{x}$.

We can now show the main theorem of this section. In the following, let us write $\mathbf{0}$ for $(-\infty, \ldots, -\infty)$ and $\mathbf{1}$ for $(0, \ldots, 0)$, so that they are indeed the units of $\oplus$ and $\otimes$ respectively.

[8] A proof of the first conjecture has actually been sketched by the author with Stéphane Gaubert. The second conjecture, to the best of this author's (and Gaubert's) knowledge, is still unproven and is left for future work. Surprisingly, while max-plus algebra has been widely studied over $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty\}$ [But10, GK07], the case of $(\overline{\mathbb{N}}, \max, +)$ does not seem to have been considered in the literature — for instance, it is not mentioned in [GP97, Table 1].

Theorem 17.8 (Complete set of unifiers for C). *Suppose Conjectures 17.1 and 17.2 and, given a problem C, let $\mathcal{W}$ and $\mathcal{A}$ be respectively the set of vectors given by the conjectures on Ψ_C and Φ_C. Then define for each $\mathbf{a} \in \mathcal{A}$ and $W \subseteq \mathcal{W}$ the substitution $\theta^{\mathbf{a}}_W$ by*[9]

$$\theta^{\mathbf{a}}_W\langle z_{\mathbf{w}}\rangle := \mathbf{w} \qquad\qquad \theta^{\mathbf{a}}_W\langle \bullet\rangle := \mathbf{1} \oplus \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in W} \mathbf{w})$$

with $\{z_{\mathbf{w}}\}_{\mathbf{w}\in W}$ fresh. Then the set of all such $\theta^{\mathbf{a}}_W$ is a complete set of unifiers for C.

Before giving the proof, let us see an example of how it can be applied.

Example 17.9. We can manually verify that the sets $\mathcal{W} = \{(0, -\infty, 0), (1, 0, -\infty)\}$ and $\mathcal{A} = \{(2, -\infty, -\infty)\}$ generate all solutions for the linear problems of Example 17.2. Therefore, Theorem 17.8 gives the following unifiers:

$$\begin{array}{rclll}
\theta^{(2,-\infty,-\infty)}_{\emptyset} & = & x_1 \mapsto 2 & x_2 \mapsto 0 & x_3 \mapsto 0 \\
\theta^{(2,-\infty,-\infty)}_{\{(0,-\infty,0)\}} & = & x_1 \mapsto 2 \sqcup z_1 & x_2 \mapsto 0 & x_3 \mapsto 0 \sqcup z_1 \\
\theta^{(2,-\infty,-\infty)}_{\{(1,0,-\infty)\}} & = & x_1 \mapsto 2 \sqcup 1 + z_2 & x_2 \mapsto 0 \sqcup z_2 & x_3 \mapsto 0 \\
\theta^{(2,-\infty,-\infty)}_{\{(0,-\infty,0),(1,0,-\infty)\}} & = & x_1 \mapsto 2 \sqcup z_1 \sqcup 1 + z_2 & x_2 \mapsto 0 \sqcup z_2 & x_3 \mapsto 0 \sqcup z_1
\end{array} \quad \square$$

Proof of Theorem 17.8. We first show that the $\theta^{\mathbf{a}}_W$ are unifiers of C. Applying Proposition 17.8, because each $\mathbf{w} \in \mathcal{W}$ is a solution to the homogeneous problem Ψ_C, it suffices to show that $\theta^{\mathbf{a}}_W\langle\bullet\rangle$ is a solution to Φ_C for each $\mathbf{a} \in \mathcal{A}$. We thus need to show that

$$\mathbf{t} \oplus \mathbf{T} \otimes (\mathbf{1} \oplus \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in W} \mathbf{w})) = \mathbf{u} \oplus \mathbf{U} \otimes (\mathbf{1} \oplus \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in W} \mathbf{w}))$$

This equation can be rearranged into

$$\mathbf{t} \oplus \mathbf{T} \otimes \mathbf{1} \oplus \mathbf{T} \otimes \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in W} \mathbf{T} \otimes \mathbf{w}) = \mathbf{u} \oplus \mathbf{U} \otimes \mathbf{1} \oplus \mathbf{U} \otimes \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in W} \mathbf{U} \otimes \mathbf{w})$$

Now note that, because each $\mathbf{w} \in W$ is a solution to Ψ_C, then $\mathbf{T} \otimes \mathbf{w}$ and $\mathbf{U} \otimes \mathbf{w}$ are equal for all $\mathbf{w} \in W$. Moreover, at the i-th line of the product $\mathbf{T} \otimes \mathbf{1}$ we have $\bigoplus_j t_i\langle x_j\rangle$, which is smaller or equal than $t_i\langle\bullet\rangle$, the i-th line of $\mathbf{t}$. So we have $\mathbf{t} \geq \mathbf{T} \otimes \mathbf{1}$, and a symmetric reasoning shows $\mathbf{u} \geq \mathbf{U} \otimes \mathbf{1}$. Therefore, to conclude now it suffices to show

$$\mathbf{t} \oplus \mathbf{T} \otimes \mathbf{a} = \mathbf{u} \oplus \mathbf{U} \otimes \mathbf{a}$$

which holds because $\mathbf{a}$ is a solution to Φ_C.

[9] We implicitly set $\theta^{\mathbf{a}}_W\langle y\rangle := -\infty$ for any other y. Note that conditions (i), (ii), (iii) from Remark 17.3 are satisfied, so this indeed defines a valid canonical form $x_i[\theta^{\mathbf{a}}_W]$ for each x_i.

Now let us show that the set of unifers made of the $\theta^{\mathbf{a}}_{W}$ is complete. Given any unifier θ, by Proposition 17.8 the vector $\theta\langle\bullet\rangle$ is a solution for Φ_C and the vector $\theta\langle y\rangle$ is a solution for Ψ_C for all $y \in \mathsf{fv}(C[\theta])$. Therefore, by Conjecture 17.1, for all $y \in \mathsf{fv}(C[\theta])$ we have

$$\theta\langle y\rangle = \bigoplus_{\mathbf{w}\in\mathcal{W}} r^{y}_{\mathbf{w}} \otimes \mathbf{w} \tag{17.1}$$

for some $\{r^{y}_{\mathbf{w}}\}_{\mathbf{w}\in\mathcal{W}}$, and by Conjectures 17.1 and 17.2 we have

$$\theta\langle\bullet\rangle = \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in\mathcal{W}} r^{\bullet}_{\mathbf{w}} \otimes \mathbf{w}) \tag{17.2}$$

for some $\mathbf{a} \in \mathcal{A}$ and $\{r^{\bullet}_{\mathbf{w}}\}_{\mathbf{w}\in\mathcal{W}}$.

Note that, because $\theta\langle\bullet\rangle \geq \theta\langle y\rangle$ for all $y \in \mathsf{fv}(C[\theta])$, we can assume $r^{\bullet}_{\mathbf{w}} \geq r^{y}_{\mathbf{w}}$, given that we can replace each $r^{\bullet}_{\mathbf{w}}$ by $\max(\{r^{\bullet}_{\mathbf{w}}\} \cup \{r^{y}_{\mathbf{w}} \mid y \in \mathsf{fv}(C[\theta])\})$ without changing the value of the right-hand side of Equation (17.2). Moreover, because $\theta\langle\bullet\rangle \geq \mathbf{1}$, we can also add a $\mathbf{1}$ to its right-hand side, once again without changing its value. We thus have

$$\theta\langle\bullet\rangle = \mathbf{1} \oplus \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in\mathcal{W}} r^{\bullet}_{\mathbf{w}} \otimes \mathbf{w}) \tag{17.3}$$

Pose $W_\bullet := \{\mathbf{w} \in \mathcal{W} \mid r^{\bullet}_{\mathbf{w}} \neq -\infty\}$ and define τ on $\{z_{\mathbf{w}}\}_{\mathbf{w}\in W_\bullet}$ by[10]

$$z_{\mathbf{w}}[\tau]\langle\bullet\rangle := r^{\bullet}_{\mathbf{w}} \qquad\qquad z_{\mathbf{w}}[\tau]\langle y\rangle := \mathsf{if}\ y \in \mathsf{fv}(C[\theta])\ \mathsf{then}\ r^{y}_{\mathbf{w}}\ \mathsf{else}\ -\infty$$

We now show that $\tau \circ \theta^{\mathbf{a}}_{W_\bullet} \simeq \theta$. First, for all $y \in \mathsf{fv}(C[\theta])$ we have

$$\begin{aligned}
(\tau \circ \theta^{\mathbf{a}}_{W_\bullet})\langle y\rangle &= \bigoplus_{\mathbf{w}\in W_\bullet} z_{\mathbf{w}}[\tau]\langle y\rangle \otimes \theta^{\mathbf{a}}_{W_\bullet}\langle z_{\mathbf{w}}\rangle && \text{by Lemma 17.2}\\
&= \bigoplus_{\mathbf{w}\in W_\bullet} r^{y}_{\mathbf{w}} \otimes \mathbf{w} && \text{by definition of } \theta^{\mathbf{a}}_{W_\bullet} \text{ and } \tau\\
&= \bigoplus_{\mathbf{w}\in\mathcal{W}} r^{y}_{\mathbf{w}} \otimes \mathbf{w} && \text{because } r^{y}_{\mathbf{w}} \leq r^{\bullet}_{\mathbf{w}} = -\infty \text{ for } \mathbf{w} \notin W_\bullet\\
&= \theta\langle y\rangle && \text{by Equation (17.1)}
\end{aligned}$$

[10] Once again, the reader can check that conditions (i), (ii) and (iii) of Remark 17.3 are verified.

Moreover, we also have

$$\begin{aligned}
(\tau \circ \theta^{\mathbf{a}}_{W_\bullet})\langle\bullet\rangle &= \theta^{\mathbf{a}}_{W_\bullet}\langle\bullet\rangle \oplus (\bigoplus_{\mathbf{w}\in W_\bullet} z_{\mathbf{w}}[\tau]\langle\bullet\rangle \otimes \theta^{\mathbf{a}}_{W_\bullet}\langle z_{\mathbf{w}}\rangle) && \text{by Lemma 17.2}\\
&= \mathbf{1} \oplus \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in W_\bullet} \mathbf{w}) \oplus (\bigoplus_{\mathbf{w}\in W_\bullet} r^\bullet_{\mathbf{w}} \otimes \mathbf{w}) && \text{by definition of } \theta^{\mathbf{a}}_{W_\bullet} \text{ and } \tau\\
&= \mathbf{1} \oplus \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in W_\bullet} r^\bullet_{\mathbf{w}} \otimes \mathbf{w}) && \text{because } \mathbf{w} \otimes r^\bullet_{\mathbf{w}} \geq \mathbf{w} \text{ for } \mathbf{w} \in W_\bullet\\
&= \mathbf{1} \oplus \mathbf{a} \oplus (\bigoplus_{\mathbf{w}\in \mathcal{W}} r^\bullet_{\mathbf{w}} \otimes \mathbf{w}) && \text{because } r^\bullet_{\mathbf{w}} = -\infty \text{ for } \mathbf{w} \notin W_\bullet\\
&= \theta\langle\bullet\rangle && \text{by Equation (17.3)}
\end{aligned}$$

concluding the proof. ■

We immediately get the following corollary.

Corollary 17.2 (The theory of predicative universe levels is finitary). *If Conjectures 17.1 and 17.2 hold, then the theory of predicative universe levels is finitary.*

Proof. By Theorem 17.8, each problem admits a finite complete set of unifiers, and by Theorem 17.1 we also know that the theory is not unitary, hence it can only be finitary. ■

One can wonder if the set Θ is also minimal. Example 17.9 actually shows that this is not the case, given that all the four unifiers are instances of $\theta^{(2,-\infty,-\infty)}_{\{(1,0,-\infty),(0,-\infty,0)\}}$ — which was expected, as this is also the m.g.u. provided by Theorem 17.5 (modulo renaming). Fortunately, if the instantiation pre-order $\preceq^X$ is decidable, a minimal complete set of unifiers can always be computed from a finite complete set by simply removing the redundant unifiers. So the only ingredient missing is the following:

Theorem 17.10 ($\preceq$ is decidable). *Given two substitutions τ, θ and a finite set of level variables X, it is decidable whether $\tau \preceq^X \theta$ holds.*

Proof. By definition of the pre-order, this holds iff there is some θ' for which we have $x[\tau][\theta'] \simeq x[\theta]$ for all $x \in X$. By Lemma 17.2, we have

$$x[\tau][\theta']\langle y\rangle = \bigoplus_{z\in\mathsf{fv}(x[\tau])} x[\tau]\langle z\rangle \otimes z[\theta']\langle y\rangle$$

and

$$x[\tau][\theta']\langle\bullet\rangle = x[\tau]\langle\bullet\rangle \oplus (\bigoplus_{z\in\mathsf{fv}(x[\tau])} x[\tau]\langle z\rangle \otimes z[\theta']\langle\bullet\rangle)$$

Writing $y_1, \dots, y_n$ for the variables in $X[\theta]$ and $z_1, \dots, z_m$ for the variables in $X[\tau]$, the problem can then be rephrased as deciding if there is $\mathbf{A} \in \overline{\mathbb{N}}_{n,m}$ and $\mathbf{a} \in \mathbb{N}_m$ solving

$$x[\theta]\langle y_i \rangle = \bigoplus_j \mathbf{A}_{i,j} \otimes x[\tau]\langle z_j \rangle$$

$$x[\theta]\langle \bullet \rangle = x[\tau]\langle \bullet \rangle \oplus (\bigoplus_j \mathbf{a}_j \otimes x[\tau]\langle z_j \rangle)$$

for all $i = 1, \dots, n$ and $x \in X$, and such that the lines of $\mathbf{A}$ are smaller than or equal to $\mathbf{a}$. It is also clear that it suffices to consider $\mathbf{A}$ for which the $\mathbf{A}_{i,j}$ are bounded by $\oplus_{x \in X} x[\theta]\langle y_i \rangle$, and $\mathbf{a}$ for which the $\mathbf{a}_j$ are bounded by $\oplus_{x \in X} x[\theta]\langle \bullet \rangle$.[11] Therefore, because the search space is finite, it follows that the problem is decidable. ■

Corollary 17.3 (m.c.u.s are computable). *Supposing Conjectures 17.1 and 17.2, we can compute a minimal set of unifers for each problem C.*

Proof. As explained above, this is a direct consequence of Theorems 17.8 and 17.10. ■

Pre-processing the unification problem

As we have seen, to calculate a m.c.u. for the problem of Example 17.2 we first calculate the set of unifiers given by Theorem 17.8, which requires us to solve the associated problems Φ_C and Ψ_C, and then minimize this set by using Theorem 17.10, revealing that the problem actually has a m.g.u. Even if we do not have the explicit complexity of our algorithm, it should be clear that for solving equations that have a m.g.u. it would be much cheaper to just apply the algorithm of Figure 17.2 instead.

Therefore, we conclude this section by proposing an optimization to our algorithm. Starting from C, we apply the rules of Figure 17.2 as much as possible, yielding some final state $\mathcal{D}; \theta$. In the case of Example 17.2 this actually solves the problem, as we obtain $\mathcal{D} = \emptyset$, but even when $\mathcal{D} \neq \emptyset$ we at least get a problem with less equations. Then, we can apply Theorem 17.8 to obtain a complete set of unifers for $\mathcal{D}$, which together with θ allows us to calculate a complete set for C. Finally, we then apply Theorem 17.10 to reduce the obtained set to a m.c.u.

Theorem 17.11 (Optimized computation of m.c.u.s). *Given a problem C, let $\mathcal{D}; \tau$ be a state of Figure 17.2 with $C; \emptyset \rightsquigarrow^* \mathcal{D}; \tau$ and, supposing Conjectures 17.1 and 17.2, let Θ be the the complete set of unifers for $\mathcal{D}$ given by Theorem 17.8. Then, by minimizing $\Theta' := \{\theta \cup \theta \circ \tau \mid \theta \in \Theta\}$ with Theorem 17.10, we obtain a m.c.u. for C.*

Proof. We only need to show that Θ' is a complete set of unifiers for C, which then trivially implies that its minimization is a m.c.u. We first show that all elements of Θ' are indeed unifiers of C. Given some $\theta \in \Theta$, we know that θ is a unifier for $\mathcal{D}$, and because $\mathrm{dom}(\tau)$

[11] There are of course smaller bounds, but here we are only worried abour decidability.

and $\mathrm{vrange}(\tau) \cup \mathrm{fv}(\mathcal{D})$ are disjoint, we have $x[\theta \cup \theta \circ \tau] = x[\tau][\theta] = x[\tau][\theta \cup \theta \circ \tau]$ for all $x \in \mathrm{dom}(\tau)$. This shows $\theta \cup \theta \circ \tau \vDash \mathcal{D}; \tau$, so by iterating Lemma 17.4 in the inverse direction we get $\theta \cup \theta \circ \tau \vDash \mathcal{C}; \emptyset$.

Now let us show that Θ' is complete. Let σ be a unifier for $\mathcal{C}$. By iterating Lemma 17.4, we get some $\sigma' =_{\mathrm{fv}(\mathcal{C})} \sigma$ with $\sigma' \vDash \mathcal{D}; \tau$. So σ' is a unifier for $\mathcal{D}$, and therefore there is $\theta \in \Theta$ and some θ' such that $x[\sigma'] \simeq x[\theta][\theta']$ for all $x \in \mathrm{fv}(\mathcal{D})$.

Moreover, looking at the unifiers given by Theorem 17.8, we see that $\mathrm{fv}(\mathcal{D}[\theta])$ only cotains fresh variables, so we can assume w.l.o.g. that $\mathrm{dom}(\theta')$ also only contains fresh variables, different from the ones in $\mathcal{D}; \tau$. We can thus define $\theta'' := \theta' \cup \sigma'$ and we have

$$x[\sigma'] \simeq x[\theta][\theta''] \tag{17.4}$$

for all $x \notin \mathrm{dom}(\theta')$: indeed, for $x \in \mathrm{fv}(\mathcal{D})$ this is automatic, and for $x \notin \mathrm{fv}(\mathcal{D})$ this follows from the fact that θ is only defined over $\mathrm{fv}(\mathcal{D})$, and thus $x[\theta][\theta''] = x[\theta''] = x[\sigma']$.

We also have $x[\sigma'] \simeq x[\tau][\sigma']$ for all $x \in \mathrm{dom}(\tau)$, and because $\mathrm{vrange}(\tau)$ is disjoint from $\mathrm{dom}(\theta')$, by Equation (17.4) we have $x[\sigma'] \simeq x[\tau][\theta][\theta''] = x[\theta \circ \tau][\theta'']$ for all $x \in \mathrm{dom}(\tau)$. Then, given that $\mathrm{dom}(\tau)$ is disjoint from $\mathrm{fv}(\mathcal{D})$, we get $x[\sigma'] \simeq x[\theta \cup \theta \circ \tau][\theta'']$ for all $x \in \mathrm{dom}(\tau) \cup \mathrm{fv}(\mathcal{D})$. Finally, we have $\mathrm{fv}(\mathcal{C}) = \mathcal{V}(\mathcal{C}; \emptyset) \subseteq \mathcal{V}(\mathcal{D}; \tau) = \mathrm{fv}(\mathcal{D}) \cup \mathrm{dom}(\tau)$ and $\sigma =_{\mathrm{fv}(\mathcal{C})} \sigma'$, thus $x[\sigma] \simeq x[\theta \cup \theta \circ \tau][\theta'']$ for all $x \in \mathrm{fv}(\mathcal{C})$, concluding the proof. ■

Chapter 18

PREDICATIVIZE: A Tool for Sharing Proofs with Predicative Systems

In this chapter we present PREDICATIVIZE, an implementation publicly available at

https://github.com/Deducteam/predicativize/

of a variant of our algorithm, and describe how it was used to translate MATITA's arithmetic library to AGDA.

18.1 The tool

Our tool is implemented on top of DKCHECK [Sai15b], a type-checker for DEDUKTI, and thus does not rely neither on the codebase of AGDA, nor on the codebase of any other proof assistant. Like in the case of UNIVERSO [Thi20], we instrument DKCHECK's conversion checker in order to implement the computation of level constraints, yielding an algorithm similar to the one of Figure 16.1.

Because the currently available type-checkers for DEDUKTI do not implement rewriting modulo for equational theories other than AC (associativity-commutativity), we used Genestier's encoding of the equational theory of universe levels [Gen20] in order to define a variant of $\mathbb{T}_{\mathbf{P}}^{\forall}$ in a DKCHECK file.

To see how everything works in practice, one can run `make running-example` which translates our running example and produces a DEDUKTI file `output/running_example.dk` and an AGDA file `agda_output/running-example.agda`. In order to test the tool with a more realistic example, the reader can also run `make test_agda`, which translates a proof of Fermat's little theorem from the DEDUKTI encoding of HOL [Thi18] to $\mathbb{T}_{\mathbf{P}}^{\forall}$.

We also note that for the moment the implementation lags behind the theory in various places, in particular by still using the older unification algorithm and the previous version of $\mathbb{T}_{\mathbf{P}}^{\forall}$ proposed in our previous work [FBB23].

In the following, let us give a high-level description of some of the practical differences with the theory presented until now.

User added constraints

As we have seen, our transformation tries to compute the most general type for a definition or declaration to be typable. However, it is not always desirable to have the most general type, as shown by the following example.

Example 18.1. Consider the local signature

$$\Phi = \mathsf{Nat} : \mathrm{Tm}\ \mathrm{U}_{\square},\ \mathsf{zero} : \mathrm{Tm}\ \mathsf{Nat},\ \mathsf{succ} : \mathrm{Tm}\ \mathsf{Nat} \longrightarrow \mathrm{Tm}\ \mathsf{Nat}$$

defining the natural numbers in $\mathbb{T}_{\mathbf{I}}$. The translation of this signature by our algorithm is

$$\Phi' = \ \mathsf{Nat} : \forall i.\mathrm{Tm}\ \mathrm{U}_i,\ \mathsf{zero} : \forall i.\mathrm{Tm}\ (\mathsf{Nat}\ i),\ \mathsf{succ} : \forall ij.\mathrm{Tm}\ (\mathsf{Nat}\ i) \longrightarrow \mathrm{Tm}\ (\mathsf{Nat}\ j)$$

However, we normally would like to impose i to be equal to j in the type of succ, or even to impose Nat not to be universe polymorphic. □

In order to solve this problem, we added to Predicativize the possibility of adding constraints by the user, in such a way that we can for instance impose Nat to be in the bottom universe, or $i = j$ in the type of the successor. Adding constraints is also used to help the unification algorithm, which can be particularly useful for simplifying unification problems when translating definitions that do not need to be universe polymorphic.

Rewrite rules

The algorithm that we presented and proved correct covers two types of entries: definitions and constants. This is enough for translating proofs written in higher-order logic or similar systems, in which every step either poses an axiom or makes a definition or proof. However, when dealing with full-fledged type theories, such as those implemented by Coq or Matita, which also feature inductive types, it is customary to use rewrite rules to encode recursion and pattern matching [Ass15, Fer21, Thi20].[1] If we simply ignore these rules when performing the translation, we would run into problems as the entries that appear after may need them to typecheck.

Therefore, our implementation extends the presented algorithm to also translate rewrite rules. In order to do this, we use DkCheck's subject reduction checker to generate constraints and proceed similarly as in the algorithm. Because this feature is still experimental, this step requires user intervention in most cases. This is done by adding new constraints over the symbols appearing in the rules, in order for their translations to

[1] Of course, ideally the definitions by pattern-matching should be compiled down to eliminators [CDP14, GMM06], and then inductive type declarations should be compiled down to W-types [Hug21] or similar constructions [CDMM10], which would allow us to work inside a fixed theory that would never need to be extended. Unfortunately, this is not what the currently available Dedukti translators do, and adding the abovementioned transformations would require some serious implementation work — though very preliminary steps in this direction have been made by the author together with Jesper Cockx, in the context of the Agda2Dk translator [Fel].

be less universe polymorphic, which helps the algorithm. This part of the translation is yet to be formally defined, and its correctness is still to be proven. Nevertheless, it has been successfully used on the translation of Matita's arithmetic library to Agda.

Agda output

Predicativize produces proofs in the theory $\mathbb{T}^{\forall}_{\mathbf{P}}$, which is a subtheory of the one implemented by the Agda proof assistant. In order to produce proofs that can be used by Agda, we also integrated in Predicativize a translator that performs a simple syntactical translation from a Dedukti file in the theory $\mathbb{T}^{\forall}_{\mathbf{P}}$ to an Agda file. For instance, `make test_agda_with_typecheck` translates Fermat's Little Theorem proof from HOL to Agda and typechecks it.

18.2 Translating Matita's arithmetic library to Agda

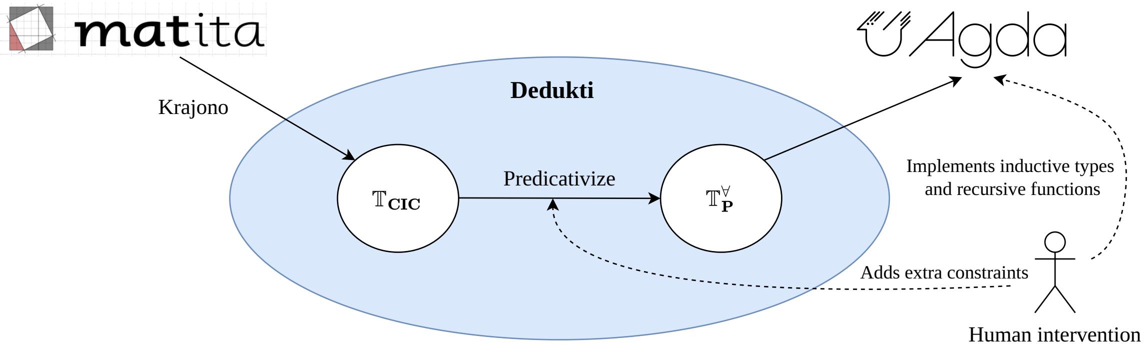

matita
Agda
Dedukti
Krajono
$\mathbb{T}_{\mathbf{CIC}}$
Predicativize
$\mathbb{T}^{\forall}_{\mathbf{P}}$
Implements inductive types and recursive functions
Adds extra constraints
Human intervention

Figure 18.1: Diagram representing the translation of Matita's arithmetic library to Agda

We now discuss how we used Predicativize to translate Matita's arithmetic library to Agda. The translation is summarized in Figure 18.1, where $\mathbb{T}_{\mathbf{CIC}}$ stands for a Dedukti theory defining the Calculus of Inductive Constructions, the underlying type theory of the Matita proof assistant.

Matita's arithmetic library [mat] was already available in Dedukti thanks to Krajono [Ass15, Ded], a translator from Matita to the theory $\mathbb{T}_{\mathbf{CIC}}$ in Dedukti. Therefore, the first step of the translation was already done for us.

Then, using Predicativize we translated the library from $\mathbb{T}_{\mathbf{CIC}}$ to $\mathbb{T}^{\forall}_{\mathbf{P}}$. As the encoding of Matita's recursive functions uses rewrite rules, their translation required some user intervention to add constraints over certain symbols, as mentioned in the previous section. Moreover, in order to help the unification algorithm, we also added constraints for fixing the levels of many definitions which were only required to be at one universe. The

	Matita	Dedukti ($\mathbb{T}_{\mathbf{CIC}}$)	Dedukti ($\mathbb{T}^{\forall}_{\mathbf{P}}$)	Agda
File size (in Kb)	67	640	570	190

Table 18.1: Comparison of (compressed) file sizes

list of all added constraints can be found in the file `extra_cstrs/matita.dk` in the implementation. These were obtained, for each of the concerned definitions, by looking at its (unconstrained) output and then adding equations involving some of its level variables — similarly to how i and j can be equated in Example 18.1. Once this step is done, the library is known to be predicative, as it typechecks in $\mathbb{T}^{\forall}_{\mathbf{P}}$.

We then used Predicativize to translate these files to Agda files. However, because the rewrite rules in the Dedukti files cannot be translated to Agda, and given that they are needed for typechecking the proofs, the library does not typecheck directly. Therefore, to finish our translation we had to define the inductive types and recursive functions manually in Agda. To do this we first assembled the type formers and constructors, which had been translated simply as postulates, into inductive type declarations. This required us to add futher constraints over some symbols, for instance between i and j in the type of the successor (Example 18.1), in order to implement them as constructors of an inductive type.

With the inductive types defined, we could then define the recursive functions (like addition), which had been translated as postulates with no computational content. Thankfully, even if we cannot translate the rewrite rules from Dedukti to Agda in a way that is accepted by Agda, we could still translate them as comments in the Agda files. Then, instead of writing such functions from scratch, we could just adapt these comments into valid Agda function declarations. We believe that this step, also needed in previous work [Thi18], could be automated by better studying the translation between different representations of recursive functions. Nevertheless, because most of Matita's arithmetic library is made of proofs, whose translation we do not need to change, automating it was not crucial in our case, so we decided to leave this study for future work.

The result of the translation is available at

`https://doi.org/10.5281/zenodo.10686897`

and, as far as we know, contains the very first proofs in Agda of Bertrand's Postulate and Fermat's Little Theorem. It also contains a variety of other interesting results such as the Binomial Law, the Chinese Remainder Theorem, and the Pigeonhole Principle. Moreover, this library typechecks with the `–safe` flag, attesting that it does not use any of Agda's more exotic and unsafe features.

We conclude by discussing some statistics about the translation. The total translation time, from Dedukti ($\mathbb{T}_{\mathbf{CIC}}$) to Dedukti ($\mathbb{T}^{\forall}_{\mathbf{P}}$) and then to Agda, is about 32 minutes on a machine with an i7 processor. We also provide in Table 18.1 a comparison of the file sizes in Matita, Dedukti (in both theories $\mathbb{T}_{\mathbf{CIC}}$ and $\mathbb{T}^{\forall}_{\mathbf{P}}$) and Agda. Here we chose to analyse their compressed sizes (using `.tar.xz`) to avoid discrepancies arising from administrative differences in the files and formats. As we see, the translation from Matita to Dedukti ($\mathbb{T}_{\mathbf{CIC}}$) increases a lot the file sizes, which are multiplied by almost 10. This is not surprising, as the original proofs are done using tactics, that are compiled to proof terms when going to Dedukti. Moreover, the representation of terms in Dedukti is much more annotated and low-level than in commonly used proof assistants, which also explains why some of this extra size is eliminated when going from Dedukti ($\mathbb{T}^{\forall}_{\mathbf{P}}$) to Agda. Yet, the proofs in Agda are still much more low-level than their Matita counterparts given that they still use proof terms instead of tactics. Finally, we see that going from Dedukti ($\mathbb{T}_{\mathbf{CIC}}$) to Dedukti ($\mathbb{T}^{\forall}_{\mathbf{P}}$) only mildly alters the file sizes, which is not surprising since our translation does not drastically change the terms.

Chapter 19

Perspectives on Proof Predicativization and Universe Level Unification

We have proposed a transformation for sharing proofs with predicative systems. Our implementation allowed to translate many non-trivial proofs from Matita's arithmetic library to Agda, showing that our proposal works well in practice.

Universe-polymorphic elaboration

Our solution is based on the use of universe-polymorphic elaboration. While elaboration algorithms are well-studied in the literature, our proposal differs from most on the use of universe level unification, which is needed in our setting for handling universe-polymorphism. Other proposals for universe-polymorphic elaboration, such as the ones by Harper and Pollack [HP91] and by Sozeau and Tabareau [ST14], avoid the use of universe level unification by allowing in their target languages for entries in the signature to come with associated sets of constraints, which are then verified locally at each use. This feature is however unfortunately not supported by Agda, the main target of our translation, which is why we need universe level unification to eliminate the constraints.

Universe level unification

Our proposal required us to study the problem of universe level unification. In order to provide an algorithm for this problem, we first contributed with a complete characterization of which equations admit a m.g.u., along with an explicit description of a m.g.u. when it exists. We then employed this characterization in the design of an constraint-postponement-based unification algorithm, which is an improvement over our preliminary work [FBB23]. It is in particular able to solve all equations that admit a m.g.u., whereas our previous algorithm was not — for instance, it was not capable of solving the first equation of Example 17.6. However some problems admitting a m.g.u. cannot be solved by our algorithm because they combine multiple equations, none of them admitting a m.g.u. (see Example 17.1).

We then proposed a second algorithm for universe level unification, which is able to compute minimal complete sets of unifiers, provided that two conjectures about max-plus algebra can be solved. Like we mentioned, we have a proof sketch of the first conjecture, however the second seems more challenging. Proving it is the main problem to be tackled in future work, which would then solve definitely the problem of universe level unification.

Our second algorithm works very similarly to Baader's unification algorithm for commutative theories [BS01], which also reduces the problem of unification to the one of solving problems in some specific algebra, and so it might come as a surprise that the theory of predicative universe levels is *not* an instance of his framework. However, even if we have not verified all details, it seems that a unification algorithm for universe levels can be obtained through a translation to the theory of *abelian idempotent monoids with an inflationary endomorphism (AIMIE)*, which *is* a commutative theory.

More precisely, AIMIE is defined by an associative, commutative and idempotent operaton $\sqcup$ with neutral element $-\infty$ and a endomorphism S which also satisfies the *inflationary* identity $\mathsf{S}\ l \sqcup l \simeq \mathsf{S}\ l$.[1] The translation is then almost the identity, except that variables i are translated as $i \sqcup 0$, and 0 is treated as an undefined constant. Then, by solving translated problems in AIMIE, it seems that we can translate-back the unifiers into the theory of universe levels, obtaining this way a m.c.u. for the original problem. This alternative algorithm would also need Conjectures 17.1 and 17.2, providing evidence that the obtained algorithm would be essentially the same as the one we gave here. Finally, if our algorithm can indeed be recovered from the one for commutative theories as it appears, then a natural question is if the strategy described above can be generalized to a class of theories, containing as a specific instance the theory of universe levels. This seems to be an interesting question that could be investigated in future work.

Regarding related work specifically on equational unification in the theory of universe levels, this problem does not appear to have been previously discussed in the literature. The proof assistant Agda features an algorithm for solving level metavariables, but to the best of our knowledge it does not seem to have been formally specified or proven correct in the literature, making it hard to provide a detailed comparison with our work. Nevertheless, practical tests suggest that our algorithm is an improvement. As an example, typechecking in Agda the entry

$$\mathsf{test} : (A : \mathsf{Set}\ \_) \to (B : \mathsf{Set}\ \_) \to (C : \mathsf{Set}\ \_) \to (R : (D : \mathsf{Set}\ \_) \to D \to D \to \mathsf{Set}\ \_) \to \mathsf{Set}\ \_$$
$$\mathsf{test} = \lambda A\ B\ C\ R \to R\ (\mathsf{Set}\ \_)\ (A \to C)\ (A \to B)$$

gives the error `Failed to solve the following constraints: _0 ⊔ _1 = _0 ⊔ _2`, however this constraint is solvable by our algorithm (see Example 17.6). Therefore, our work could also be used to improve Agda's unification algorithm.

[1] Note that this theory is inadequate to be used in the context of type universes, as the equation $\mathsf{S}\ {-\infty} \simeq -\infty$ implies $\mathsf{U}_{-\infty} : \mathsf{Tm}\ \mathsf{U}_{-\infty}$, yielding an inconsistent theory.

Predicativize

For future work, we would also like to look at possible ways of making Predicativize less dependent on user intervention. In particular, the translation of inductive types and recursive functions involves some considerable manual work. We thus expect improvements in this direction to be needed in order to translate larger proof libraries.

Another problem that we have also not tackled is the one of *concept alignment*, that is, adapting the translated statements and proofs so that they use the definitions from the Agda standard library, which would make our proofs more usable for Agda users. This is already being investigated in the context of the translation from HOL-Light to Coq by Blanqui [Bla24], however his proposed method requires a lot of manual invervention. One can thus wonder if a more automatic solution could be applied. The use of proof transfer tools such as Trocq [CCM24] could be an interesting direction to explore.

Bibliography

[AA11] Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for type: Type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011.

[Abe13] Andreas Abel. Normalization by evaluation: Dependent types and impredicativity. *Habilitation. Ludwig-Maximilians-Universität München*, 2013.

[AC05] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for martin-löf's logical framework with surjective pairs. In *Typed Lambda Calculi and Applications: 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005. Proceedings 7*, pages 23–38. Springer, 2005.

[ACP11] Andreas Abel, Thierry Coquand, and Miguel Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. *Logical Methods in Computer Science*, 7, 2011.

[Acz78] Peter Aczel. A General Church-Rosser Theorem. Unpublished note, 1978.

[Ada06] Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, 2006.

[Ada08] Robin Adams. Lambda-free logical frameworks, 2008. https://arxiv.org/abs/0804.1879.

[ADJL17] Ali Assaf, Gilles Dowek, Jean-Pierre Jouannaud, and Jiaxiang Liu. Untyped Confluence In Dependent Type Theories. working paper or preprint, April 2017.

[AK16a] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *SIGPLAN Not.*, 51(1):18–29, jan 2016.

[AK16b] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 18–29, New York, NY, USA, 2016. Association for Computing Machinery.

[AKvV23] Thorsten Altenkirch, Ambrus Kaposi, Artjoms Šinkarovs, and Tamás Végh. The Münchhausen Method in Type Theory. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, 2007.

[AOV18] Andreas Abel, Joakim Ohman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, January 2018.

[AR12] Andrea Asperti and Wilmer Ricciotti. A proof of bertrand's postulate. *Journal of Formalized Reasoning*, 5(1):37–57, 2012.

[Ass14] Ali Assaf. A calculus of constructions with explicit subtyping. In *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39, 2014.

[Ass15] Ali Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École polytechnique, September 2015.

[Atk18] Robert Atkey. Simplified observational type theory, 2018. Implementation at https://github.com/bobatkey/sott.

[AVW17] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. Normalization by evaluation for sized dependent types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–30, 2017.

[BA24] Christoph Benzmüller and Peter Andrews. Church's Type Theory. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2024 edition, 2024.

[Bar91] Henk Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1:125–154, 04 1991.

[Bar92] Henk Barendregt. *Lambda Calculi with Types*, volume 2, pages 117–309. 01 1992.

[Bar98] Gilles Barthe. The relevance of proof-irrelevance. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 755–768, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[BB88] Franz Baader and Wolfram Büttner. Unification in commutative idempotent monoids. *Theoretical Computer Science*, 56(3):345–353, 1988.

[BC22] Marc Bezem and Thierry Coquand. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theoretical Computer Science*, 913:1–7, 2022.

[BCDE23] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. Type Theory with Explicit Universe Polymorphism. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BDG$^+$23] Frédéric Blanqui, Gilles Dowek, Emilie Grienenberger, Gabriel Hondet, and François Thiré. A modular construction of type theories. *Logical Methods in Computer Science*, Volume 19, Issue 1, February 2023.

[BDS13] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types.* Cambridge University Press, 2013.

[BFG97] Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.

[BGH19] Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:21, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[BK22] Andrej Bauer and Anja Petkovic Komel. An extensible equality checking algorithm for dependent type theories. *Log. Methods Comput. Sci.*, 18(1), 2022.

[BKdVT03] Marc Bezem, Jan Willem Klop, Roel de Vrijer, and Terese. *Term Rewriting Systems.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003.

[Bla01] Frédéric Blanqui. *Théorie des types et réécriture. (Type theory and rewriting).* PhD thesis, University of Paris-Sud, Orsay, France, 2001.

[Bla03] F. Blanqui. Rewriting modulo in deduction modulo. In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 2706, 2003. 15 pages.

[Bla05] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

[Bla20] Frédéric Blanqui. Type safety of rewrite rules in dependent types. In *5th International Conference on Formal Structures for Computation and Deduction*, 2020.

[Bla22] Frédéric Blanqui. Encoding type universes without using matching modulo AC. In *Proceedings of the 7th International Conference on Formal Structures for Computation and Deduction*, Leibniz International Proceedings in Informatics 228, 2022.

[Bla24] Frédéric Blanqui. Translating hol-light proofs to coq. In *Proceedings of 25th Conference on Logic for Pro*, volume 100, pages 1–18, 2024.

[BM21] Bruno Barras and Valentin Maestracci. Implementation of two layers type theory in dedukti and application to cubical type theory. *Electronic Proceedings in Theoretical Computer Science*, 332:54–67, jan 2021.

[BNW19] Michael Beeson, Julien Narboux, and Freek Wiedijk. Proof-checking Euclid. *Annals of Mathematics and Artificial Intelligence*, page 53, January 2019.

[BS00] Gilles Barthe and Morten Heine Sørensen. Domain-free pure type systems. *Journal of functional programming*, 10(5):417–452, 2000.

[BS01] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[But10] Peter Butkovič. *Max-linear systems: theory and algorithms*. Springer Science & Business Media, 2010.

[BvR97] Gilles Barthe and Femke van Raamsdonk. Termination of algebraic type systems: The syntactic approach. In Michael Hanus, Jan Heering, and Karl Meinke, editors, *Algebraic and Logic Programming*, pages 174–193, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Car86] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[CCHM18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[CCM24] Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 239–268, Cham, 2024. Springer Nature Switzerland.

[CD07] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[CDMM10] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. *ACM Sigplan Notices*, 45(9):3–14, 2010.

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without k. *SIGPLAN Not.*, 49(9):257–268, aug 2014.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[CK24] Liang-Ting Chen and Hsiang-Shang Ko. A formal treatment of bidirectional typing. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 115–142, Cham, 2024. Springer Nature Switzerland.

[Coq86] Thierry Coquand. An analysis of Girard's paradox. Technical Report RR-0531, INRIA, May 1986.

[Coq96] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1-3):167–177, 1996.

[Coq13] Thierry Coquand. Presheaf model of type theory. Unpublished note available at `http://www.cse.chalmers.se/~coquand/presheaf.pdf`, 2013.

[dB94] Nicolaas Govert de Bruijn. A survey of the project automath. In R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 141–161. Elsevier, 1994.

[Ded] Deducteam. Matita's arithmetic library in Dedukti. `https://github.com/Deducteam/Deducteam.github.io/blob/master/data/libraries/matita.tar.gz`.

[Del20] Tristan Delort. Importer les preuves de Logipedia dans Agda. Internship report, Inria Saclay Ile de France, November 2020.

[DHKP96] Gilles Dowek, Thérese Hardin, Claude Kirchner, and Frank Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In *JICSLP*, pages 259–273, 1996.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys (CSUR)*, 54(5):1–38, 2021.

[Dow93] Gilles Dowek. The undecidability of typability in the lambda-pi-calculus. In *International Conference on Typed Lambda Calculi and Applications*, pages 139–145. Springer, 1993.

[Dow17] Gilles Dowek. Models and Termination of Proof Reduction in the lambda Pi-Calculus Modulo Theory. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 109:1–109:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[DP04] Jana Dunfield and Frank Pfenning. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 281–292, New York, NY, USA, 2004. Association for Computing Machinery.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction–recursion. *The Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.

[Dyb94] Peter Dybjer. Inductive families. *Formal aspects of computing*, 6:440–465, 1994.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The journal of symbolic logic*, 65(2):525–549, 2000.

[FBB23] Thiago Felicissimo, Frédéric Blanqui, and Ashish Kumar Barnawal. Translating Proofs from an Impredicative Type System to a Predicative One. In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, volume 252 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Fel] Thiago Felicissimo. Compiling dependent pattern matching to elimination principles in Dedukti (STSM report). https://europroofnet.github.io/_pages/stsm/felicissimo-rep.pdf.

[Fel22a] Thiago Felicissimo. Adequate and Computational Encodings in the Logical Framework Dedukti. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:18, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Fel22b] Thiago Felicissimo. No need to be implicit! Unpublished note at `https://github.com/thiagofelicissimo/my-files/blob/master/pts-epts.pdf?raw=true`, 2022.

[Fel24a] Thiago Felicissimo. Generic bidirectional typing for dependent type theories. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 143–170, Cham, 2024. Springer Nature Switzerland.

[Fel24b] Thiago Felicissimo. Second-order church-rosser modulo, without normalization. In *13th International Workshop on Confluence*, 2024.

[Fer21] Gaspard Ferey. *Higher-Order Confluence and Universe Embedding in the Logical Framework*. These, Université Paris-Saclay, June 2021.

[FPT99] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 193–202. IEEE, 1999.

[FW24] Thiago Felicissimo and Théo Winterhalter. Impredicativity, Cumulativity and Product Covariance in the Logical Framework Dedukti. working paper or preprint, February 2024.

[G$^+$08] Georges Gonthier et al. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[Gen20] Guillaume Genestier. Encoding agda programs using rewriting. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 31:1–31:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Geu92] Herman Geuvers. The church-rosser property for beta eta -reduction in typed lambda -calculi. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, 1992.

[Geu93] Herman Geuvers. *Logics and type systems*. PhD thesis, University of Nijmegen, 1993.

[Geu95] Herman Geuvers. The Calculus of Constructions and Higher Order Logic. In Ph. de Groote, editor, *The Curry-Howard isomorphism*, volume 8 of *Cahiers du Centre de logique*, pages 139–191. Université catholique de Louvain, 1995.

[GK07] Stéphane Gaubert and Ricardo D. Katz. The minkowski theorem for max-plus convex sets. *Linear Algebra and its Applications*, 421(2–3):356–369, March 2007.

[GKMW10] Herman Geuvers, Robbert Krebbers, James McKinna, and Freek Wiedijk. Pure type systems without explicit contexts. In Karl Crary and Marino Miculan, editors, *Proceedings 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2010, Edinburgh, UK, 14th July 2010*, volume 34 of *EPTCS*, pages 53–67, 2010.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. *Eliminating Dependent Pattern Matching*, pages 521–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[GN91] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

[GP97] Stéphane Gaubert and Max Plus. Methods and applications of (max,+) linear algebra. In Rüdiger Reischuk and Michel Morvan, editors, *STACS 97*, pages 261–282, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[GPS99] Harald Ganzinger, Frank Pfenning, and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Automated Deduction—CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings 16*, pages 202–206. Springer, 1999.

[Gra23] Daniel Gratzer. *Syntax and semantics of modal type theory*. PhD thesis, Aarhus University, 2023.

[Gri23] Emilie Grienenberger. Expressing Ecumenical Systems in the lambdaPi-Calculus Modulo Theory. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, volume 269 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[GSA$^+$22] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. Controlling unfolding in type theory. *arXiv preprint arXiv:2210.05420*, 2022.

[GSB19] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*, volume 7. Cambridge university press Cambridge, 1989.

[Gé] Yoan Géran. Euclid's elements book 1 in dedukti. `https://github.com/Karnaj/sttfa_geocoq_euclid`.

[Ham22] Makoto Hamana. Complete algebraic semantics for second-order rewriting systems based on abstract syntax with variable binding. *Mathematical Structures in Computer Science*, 32(4):542–573, 2022.

[Har21a] Robert Harper. An equational logical framework for type theories. *arXiv preprint arXiv:2106.01484*, 2021.

[Har21b] Robert Harper. Notes on logical frameworks. Unpublished note at `https://www.cs.cmu.edu/~rwh/papers/lfias/lf.pdf`, 2021.

[HB20] Gabriel Hondet and Frédéric Blanqui. The new rewriting engine of dedukti (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 35:1–35:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[HB21] Gabriel Hondet and Frédéric Blanqui. Encoding of Predicate Subtyping with Proof Irrelevance in the Lambdapi-Calculus Modulo Theory. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[HB23] Philipp G. Haselwarter and Andrej Bauer. Finitary type theories with and without contexts. *J. Autom. Reason.*, 67(4), oct 2023.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, jan 1993.

[HL07] Robert Harper and Daniel R Licata. Mechanizing metatheory in a logical framework. *Journal of functional programming*, 17(4-5):613–673, 2007.

[Hof97] Martin Hofmann. Syntax and semantics of dependent types. *Extensional Constructs in Intensional Type Theory*, pages 13–54, 1997.

[HP91] Robert Harper and Robert Pollack. Type checking with universes. *Theor. Comput. Sci.*, 89(1):107–136, aug 1991.

[HP05] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the lf type theory. *ACM Transactions on Computational Logic (TOCL)*, 6(1):61–101, 2005.

[Hue80] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM (JACM)*, 27(4):797–821, 1980.

[Hug21] Jasper Hugunin. Why not w? In *26th International Conference on Types for Proofs and Programs (TYPES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[JK84] Jean-Pierre Jouannaud and Hélene Kirchner. Completion of a set of rules modulo a set of equations. In *11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1984.

[KHS19] Ambrus Kaposi, Simon Huber, and Christian Sattler. Gluing for type theory. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[KKA19] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[Kov22] András Kovács. Generalized Universe Hierarchies and First-Class Universe Levels. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*, volume 216 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:17, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Kov23] András Kovács. elaboration-zoo. `https://github.com/AndrasKovacs/elaboration-zoo`, 2023.

[Kvv93] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1):279–308, 1993.

[LB21] Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume

193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[LB22] Meven Lennon-Bertrand. *Bidirectional Typing for the Calculus of Inductive Constructions*. PhD thesis, Nantes Université, 2022.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. *SIGPLAN Not.*, 42(1):173–184, jan 2007.

[mat] Matita's arithmetic library. https://github.com/LPCIC/matita/tree/master/matita/matita/lib/arithmetics.

[McB00] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 2000.

[McB18] Conor McBride. Basics of bidirectionalism. https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/, 2018.

[McB22] Conor McBride. Types who say ni. In preparation, draft available at https://github.com/pigworker/TypesWhoSayNi/, 2022.

[MH94] Aart Middeldorp and Erik Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of logic and computation*, 1(4):497–536, 1991.

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.

[ML84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

[ML94] Per Martin-Löf. *Analytic and Synthetic Judgements in Type Theory*, pages 87–99. Springer Netherlands, Dordrecht, 1994.

[MMS$^+$21] Kenji Maillard, Nicolas Margulies, Matthieu Sozeau, Nicolas Tabareau, and Éric Tanter. The multiverse: Logical modularity for proof assistants. *CoRR*, abs/2108.10259, 2021.

[MN98] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theoretical computer science*, 192(1):3–29, 1998.

[MW96] Paul-André Mellies and Benjamin Werner. A generic normalisation proof for pure type systems. In *International Workshop on Types for Proofs and Programs*, pages 254–276. Springer, 1996.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Nor13] Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

[NPP08] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):1–49, 2008.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Lo¨f's type theory: an introduction*. Clarendon Press, USA, 1990.

[PD10] Brigitte Pientka and Jana Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*, pages 15–21. Springer, 2010.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[Pfe01a] Frank Pfenning. *Computation and Deduction*. 2001. In preparation, draft available at https://www.iro.umontreal.ca/~monnier/6172/cd.pdf.

[Pfe01b] Frank Pfenning. Logical frameworks. *Handbook of automated reasoning*, 2:1063–1147, 2001.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 328–345, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Pol92] Randy Pollack. Typechecking in pure type systems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, B astad, Sweden*, pages 271–288, 1992.

[Pre94] Christian Prehofer. Higher-order narrowing. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516. IEEE, 1994.

[PT00] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[PT18] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 245–271, Cham, 2018. Springer International Publishing.

[PT22] Loïc Pujet and Nicolas Tabareau. Observational equality: now for good. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–27, 2022.

[PT24] Loïc Pujet and Nicolas Tabareau. Observational equality meets cic. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 275–301, Cham, 2024. Springer Nature Switzerland.

[Ree08] Jason Reed. Redundancy elimination for lf. *Electronic Notes in Theoretical Computer Science*, 199:89–106, 2008.

[Ree09] Jason Reed. Higher-order constraint simplification in dependent type theory. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, pages 49–56, 2009.

[Sai15a] Ronan Saillard. Rewriting modulo β in the $\lambda\pi$-calculus modulo. *Electronic Proceedings in Theoretical Computer Science*, 185:87–101, July 2015.

[Sai15b] Ronan Saillard. *Type checking in the Lambda-Pi-calculus modulo: theory and practice*. PhD thesis, Mines ParisTech, France, 2015.

[SH12] Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, 2012.

[ST14] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In *International Conference on Interactive Theorem Proving*, pages 499–514. Springer, 2014.

[Ste19] Jonathan Sterling. Algebraic type theory and universe hierarchies. *arXiv preprint arXiv:1902.08848*, 2019.

[Ste22a] Jon Sterling. How to code your own type theory, 2022. https://www.youtube.com/watch?v=DEj-_k2Nx6o.

[Ste22b] Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, 2022.

[Str93] Thomas Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, page 57, 1993.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[Tea] Agda Development Team. Agda 2.6.2.1 documentation. https://agda.readthedocs.io/en/v2.6.2.1/index.html.

[Thi18] François Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. In Frédéric Blanqui and Giselle Reis, editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*, volume 274 of *EPTCS*, pages 57–71, 2018.

[Thi20] François Thiré. *Interoperability between proof systems using the logical framework Dedukti*. PhD thesis, ENS Paris-Saclay, 2020.

[Uem21] Taichi Uemura. *Abstract and concrete type theories*. PhD thesis, University of Amsterdam, 2021.

[vO99] Vincent van Oostrom. Normalisation in weakly orthogonal rewriting. In *International Conference on Rewriting Techniques and Applications*, pages 60–74. Springer, 1999.

[Voe14] Vladimir Voevodsky. A universe polymorphic type system, 2014. Unfinished manuscript.

[vOvR94] Vincent van Oostrom and Femke van Raamsdonk. Weak orthogonality implies confluence: The higher-order case. In Gerhard Goos, Juris Hartmanis, Anil Nerode, and Yu. V. Matiyasevich, editors, *Logical Foundations of Computer Science*, volume 813, pages 379–392. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. Series Title: Lecture Notes in Computer Science.

[vR97] Femke van Raamsdonk. Outermost-fair rewriting. In *International Conference on Typed Lambda Calculi and Applications*, pages 284–299. Springer, 1997.

[WCPW03] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework i: Judgments and properties. Technical report, Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2003.

[ZS17] Beta Ziliani and Matthieu Sozeau. A comprehensible guide to a new unifier for cic including universe polymorphism and overloading. *Journal of Functional Programming*, 27:e10, 2017.