KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Dependent pattern matching and proof-relevant unification

Jesper Cockx

Supervisors:
Prof. dr. ir. F. Piessens
Dr. D. Devriese

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2017

Dependent pattern matching and proof-relevant unification

Jesper COCKX

Examination committee:
Prof. dr. ir. H. Hens, chair
Prof. dr. ir. F. Piessens, supervisor
Dr. D. Devriese, supervisor
Prof. dr. ir. G. Janssens
Prof. dr. ir. B. Jacobs
Prof. dr. ir. T. Schrijvers
Prof. dr. P. Wadler
(University of Edinburgh)
Prof. dr. A. Abel
(Chalmers University)

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

June 2017

Preface

Acknowledgements. Thank you to my promotors Frank and Dominique for your advice and constant support. This thesis would never have existed without you, obviously.

Thank you to the jury members who did an excellent job at reading everything in this thesis critically and providing very useful feedback. It may be a cliché, but I think the text really improved a lot thanks to you.

Thank you to the FWO for giving me the opportunity to work on this fascinating subject during these four years.

Thank you to Andreas, Ulf, Nils, Andrea, Guillaume, James, Víctor, Fredrik, and all the other Agda developers and regular attendants of the Agda Implementor's Meeting. These meetings are always a lot of fun and were a great source of inspiration for this thesis.

Thank you to my office mates Mathy and Raoul, for respectively not hacking my devices[1] and coming out of your enclave occasionally[2]. We certainly had a lot of fun and interesting discussions during these four years. Just try not to quarrel too much when not there.

Of course also thanks to everyone in our lunch group (past and present): Frédéric, Milica, Pieter, Rula, Jan Tobias, Marco, Raoul, Mathy, Gitte, Neline, Andreas, Jo, Sven, Dimitri, Kobe, Stelios, Lau, and Vera. Our mission to make lunch great again was a big triumph.

Thank you to the secretaries and the DistriNet business office for helping me with all kinds of practical matters and making the department such a nice place to work.

[1] Unless you did hack them and you didn't tell me.

[2] Also thanks for the moelleux.

The members of Visionary Investments deserve my sincere gratitude for the excellent coffee. I don't know whether I would have survived four years on bad coffee. Jonathan also deserves special thanks for all the interesting coffee breaks during the day and whisky tastings at night.

Thanks also to Kristof, Rinde, Emad, Tung, and Jonathan for the fun at the after-work LAN parties. When do you have time for another one?

Last and perhaps most of all, thanks to my parents for their love, support, encouragement, and help during the last four years as well as all the years before. You're the best.

Abstract

Dependent type theory is a powerful language for writing functional programs with very precise types. It is used to write not only programs but also mathematical proofs that these programs satisfy certain properties. Because of this, languages based on dependent types – such as Coq, Agda, and Idris – are used both as programming languages and as interactive proof assistants.

While dependent types give strong guarantees about your programs and proofs, they also impose equally strong requirements on them. This often makes it harder to write programs in a dependently typed language compared to one with a simpler type system. For this reason certain techniques have been developed, such as dependent pattern matching and specialization by unification. These techniques provide an intuitive way to write programs and proofs in dependently typed languages.

Previously, dependent pattern matching had only been shown to work in a limited setting. In particular, it relied on the K axiom – also known as the uniqueness of identity proofs – to remove equations of the form $x = x$. This axiom is inadmissible in many type theories, particularly in the new and promising branch known as homotopy type theory (HoTT). As a result, programs and proofs in these new theories cannot make use of dependent pattern matching and are as a result much harder to write, modify, and understand. Additionally, the interaction of dependent pattern matching with small but practical features such as eta-equality for record types and postponing of unification constraints was poorly understood, resulting in subtle bugs and inconsistencies.

In this thesis, we develop dependent pattern matching and unification in a general setting that does not require the K axiom, both from a theoretical perspective and a practical one. In particular, we present a proof-relevant unification algorithm, where each unification rule produces evidence of its correctness. This evidence guarantees that all unification rules are correct by

construction, and also gives a computational characterization to each unification rule.

To ensure that these techniques are sound and will stay so in face of future extensions to type theory, we show how to translate them to more basic primitive constructs, i.e. the standard datatype eliminators. During this translation, we pay special attention to the computational content of all constructions involved. This guarantees that the intuitions from regular pattern matching carry over to a dependently typed setting.

Based on our work, we implemented a complete overhaul of the algorithm for checking definitions by dependent pattern matching in Agda. Our new implementation fixes a substantial number of issues in the old implementation, and is at the same time less restrictive than the old ad-hoc restrictions. Thus it puts the whole system back on a strong foundation. In addition, our work has already been used as the basis for other implementations of dependent pattern matching, such as the Equations package for Coq and the Lean theorem prover.

The work in this thesis eliminates all implicit assumptions introduced to the type theory by pattern matching and unification. In the future, we may also want to integrate new principles with pattern matching, for example the higher inductive types introduced by HoTT. The framework presented in this thesis also provides a solid basis for such extensions to be built on.

Beknopte samenvatting

Afhankelijke typetheorie is een krachtige taal voor het schrijven van functionele programma's met zeer precieze types. In deze taal kunnen niet alleen programma's geschreven worden, maar ook wiskundige bewijzen dat deze programma's voldoen aan bepaalde eigenschappen. Om deze reden worden talen gebaseerd op typetheorie – zoals Coq, Agda, en Idris – zowel gebruikt als programmeertalen alsook als interactieve bewijsassistenten.

Hoewel afhankelijke types sterke garanties geven over programma's en bewijzen, stellen ze evenzeer zware eisen in het gebruik. Dit maakt het dikwijls moeilijker om programma's te schrijven in een afhankelijk getypeerde taal dan in een taal met een eenvoudiger typesysteem. Om deze reden zijn bepaalde technieken ontwikkeld, zoals *afhankelijke patroonherkenning* en *specialisatie door middel van unificatie*. Deze technieken maken het veel intuïtiever om programma's en bewijzen te schrijven in afhankelijk getypeerde talen.

Afhankelijke patroonherkenning was voorheen echter niet algemeen toepasbaar. Het hing met name af van het K-axioma – ook bekend als de uniciteit van identiteitsbewijzen – om vergelijkingen van de vorm $x = x$ af te handelen. Dit axioma is ontoelaatbaar in vele varianten van typetheorie, in het bijzonder in een nieuwe en veelbelovende stroming genaamd *homotopietypetheorie* (HoTT). Bijgevolg kunnen programma's en bewijzen in zulke varianten geen gebruik maken van afhankelijke patroonherkenning, waardoor ze moeilijker zijn om te schrijven, aan te passen, en te begrijpen. Bovendien begreep men onvoldoende de interactie tussen afhankelijke patroonherkenning en kleine maar praktische eigenschappen zoals eta-gelijkheid voor record types en uitgestelde unificatieproblemen, wat resulteerde in subtiele fouten en inconsistenties.

In deze thesis ontwikkelen we afhankelijke patroonherkenning en unificatie zowel vanuit een theoretisch als een praktisch perspectief, zonder te vertrouwen op het K-axioma. We geven hiervoor een *bewijs-relevant* unificatiealgoritme, waarbij elke unificatieregel een getuigenis produceert van zijn eigen correctheid. Deze

getuigenis garandeert dat alle unificatieregels correct zijn *per constructie*, en geeft daarnaast een computationele interpretatie aan elke unificatieregel.

Om ervoor te zorgen dat deze technieken consistent zijn en dat ook zullen blijven bij toekomstige uitbreidingen, tonen we hoe deze vertaald kunnen worden naar de basisprincipes van de typetheorie, i.e. de standaard *datatype-eliminatoren*. In de loop van deze vertaling besteden we bijzondere aandacht aan de computationele inhoud van elke constructie. Op deze manier garanderen we dat de intuïties uit reguliere patroonherkenning overgedragen kunnen worden naar afhankelijke patroonherkenning.

Op basis van ons werk geven we een volledig nieuwe implementatie van het unificatiealgoritme dat Agda gebruikt om definities met afhankelijke patroonherkenning te controleren. Onze nieuwe implementatie lost een substantieel aantal problemen op in de oude implementatie, en is bovendien minder restrictief dan de oude ad-hoc beperkingen. Ons werk is daarnaast reeds gebruikt als de basis voor andere implementaties van afhankelijke patroonherkenning, zoals het Equations-pakket voor Coq en de bewijsassistent Lean.

Het werk in deze thesis elimineert alle impliciete veronderstellingen die geïntroduceerd werden door patroonherkenning en unificatie. In de toekomst willen we ook nieuwe principes integreren met patroonherkenning, zoals bijvoorbeeld *hoger-inductieve types* uit HoTT. Deze thesis geeft een solide fundament om zulke uitbreidingen op te bouwen.

Contents

Chapter 1

Introduction

> *The Wheel of Time turns, and Ages come and pass, leaving memories that become legend. Legend fades to myth, and even myth is long forgotten when the Age that gave it birth comes again. In one Age, called the Third Age by some, an Age yet to come, an Age long past, a wind rose in the Mountains of Mist. The wind was not the beginning, there are neither Beginnings nor Endings to the turning of the Wheel of Time. But it was* a *beginning.*
>
> — Robert Jordan (1990)

Programming is hard. This is proven again and again by the numerous bugs, vulnerabilities, and other weird behaviours in the software we use every day. Tests, code reviews and better methodologies can catch some of these problems, but never eliminate all of them. Languages based on dependent type theory, such as Coq (The Coq development team, 2016), Agda (The Agda development team, 2016) and Idris (The Idris community, 2017), promise a way out of this mess by allowing us to write proofs that a program matches its specification. The typechecker can check these proofs automatically, thus guaranteeing with absolute certainty that the program works as intended.

Compared to other tools for program verification, dependently typed languages are unique in the fact that they combine both programs and proofs in the same language. This means the same language can be used both as a *programming language* and as an interactive *proof assistant*. In particular, the same techniques that are used for writing programs can also be used for writing proofs about these programs.

Dependent pattern matching is a powerful example of such a technique for writing programs and proofs in dependently typed languages (Coquand, 1992). It combines the intuitive notation of pattern matching from functional languages such as Haskell and ML with the mathematical techniques of case analysis and induction. Concretely, it allows you to define a function simply by stating a number of equalities that it should satisfy. Hence the computational meaning of a function defined by dependent pattern matching is immediately obvious from its definition.

Compared to other, more primitive notions to define functions in dependent type theory, dependent pattern matching automates many steps that can be performed purely mechanically. In particular, during case analysis there are certain equations that have to be solved in each subcase. Dependent pattern matching relies on a unification algorithm to solve these equations automatically. This means definitions by dependent pattern matching are often much shorter and easier to read compared to the more primitive techniques.

However, the original formulation of dependent pattern matching by Coquand only works in a limited setting. In particular, it is incompatible with new versions of dependent type theory such as homotopy type theory (HoTT) (The Univalent Foundations Program, 2013). The source of this limitation lies in the unification algorithm that is employed for case splitting, which relies implicitly on two axioms – the K axiom and the injectivity of type constructors – that are incompatible with HoTT.

This thesis removes all implicit assumptions from dependent pattern matching and puts it on a solid theoretical basis. To do so, we[1] present a *proof-relevant* unification algorithm that produces for each unifier it produces an accompanying proof of its correctness. In this chapter, we introduce the main topics of this thesis. Section 1.1 gives a general overview of the setting: type theory and dependent types. Section 1.2 then zooms in on dependent pattern matching and specialization by unification. Section 1.3 highlights some of the problems that occur when using the original formulation of dependent pattern matching together with HoTT. To get to the root of this problem, Section 1.4 show how to translate definitions by pattern matching to the more basic datatype eliminators. Section 1.5 rounds out the introduction by discussing three recurring themes of this thesis: proof relevance, axiomatic freedom, and computational behaviour. Finally, Section 1.6 gives an overview of the rest of the thesis and lists its main technical contributions.

[1] Throughout this thesis, I use 'I' only for the Contributions section; elsewhere I prefer to use 'we'. If you want, you can think of it as an adventure we go on together.

1.1 Type theory

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*
>
> — Donald Knuth (1977)

A *type system* is a discipline imposed on computer programs to rule out some class of programs that don't make sense while keeping those that do. For example, a type system ideally allows you to write $1 + 1$, but will yell at you if you write something like $1 +$ "one". It does so by assigning types to certain parts of a program, such as `Int` for integers or `String` for strings.

If some program part a has type T, then it is written:

$$a : T \tag{1.1}$$

For example, 1 : `Int` and "one" : `String`.

Type systems are used in more programming languages and in more varieties than would be feasible to name here. One thing they have in common is that they are conservative: they always rule out some 'good' programs together with the bad ones. One of the main driving forces behind the development of new type systems is then to increase their power so they recognize more 'good' programs while still ruling out the bad ones.

At this point, it may seem that type systems are only there to put increasingly strict requirements on what programs you're allowed to write. This is indeed why type systems were added to programming languages in the first place: to prevent bugs. However, people have discovered that type systems are good for more than just ruling out errors from their programs:

- Types serve as documentation to your programs, describing the meaning and intention of each piece of code.
- Types give a clear interface to your programs, allowing you to easily compose software components.
- Types restrict the possible inputs of a program, so the compiler can optimize your code more heavily for those specific inputs.
- Types give more structure to your programs, so programming environments can use this structure to help you with refactoring a piece of code.

All of these goals benefit from having expressive types that give as much information as possible about their terms.

1.1.1 The simply typed lambda calculus

Even before the first programming language was invented, type systems were already studied by mathematicians. One of the first of such systems is the simply typed lambda calculus (STLC) by Church (1940), which he used to rule out the paradoxes in the untyped lambda calculus. Since then, STLC has been used by computer scientists as a simple model of a typed functional programming language. Because it is so easy to extend, it has been used as a basis for studying many other type systems, including the dependent type theory used in this thesis.

The main power of STLC comes from its simplicity. It has only one way to construct types: the function type $A \to B$, where A and B can be arbitrary types (including other function types). If we have a function $f : A \to B$ and some value $u : A$, then we can apply f to u to get $f\ u : B$. Conversely, if we have some $v : B$ in which there may occur some variable $x : A$, then we can construct the lambda-abstraction $\lambda x.\ v : A \to B$.

Any function in STLC is necessarily *total*: it is defined for all possible inputs and evaluating it is guaranteed to terminate. It is possible to waive this restriction by introducing a fixpoint combinator. In this thesis (and in dependent type theory in general) we keep this restriction and instead extend the language in different ways while keeping the guarantee that all functions are total.

Usually, STLC is extended with one or more base types, such as the type `Bool` of booleans `true` and `false` and the type $\mathbb{N}$ of unary natural numbers `zero`, `suc zero`, `suc` (`suc zero`), ... Other examples are the unit type $\top$ consisting of the single element `tt` and the empty type $\bot$ with no elements at all (whose use will become apparent in the next section). STLC can also be extended with other type constructors besides the function arrow, such as the product type $A \times B$ consisting of pairs (x, y) where $x : A$ and $y : B$, and the sum type $A \uplus B$ consisting of elements `left` x for $x : A$ and `right` y for $y : B$.

All these types are examples of *(algebraic) datatypes*. A datatype is a type inductively defined by a number of constructors. For example, `Bool` is defined by the two constructors `true` and `false`, and $\mathbb{N}$ has the two constructors `zero` and `suc`. The fact that datatypes are defined inductively is what makes it possible to do pattern matching on them (Section 1.2).

1.1.2 Propositions as Types

Curry (1934) and Howard (1969) noted a curious correspondence between functions in the simply typed lambda calculus and implication in (intuitionistic)

propositional logic: types can be seen as propositions, and an element of a type can be seen as a *proof* of the corresponding proposition. Under this correspondence, a function type $A \to B$ corresponds to an implication $A \Rightarrow B$, and any function $f : A \to B$ corresponds to a proof of the implication $A \Rightarrow B$. Indeed, a function from A to B gives us exactly a way to transform an arbitrary proof of A into a proof of B. Moreover, there is a similar correspondence between other types and propositions:

- The product type $A \times B$ corresponds to the conjunction $A \wedge B$.
- The sum type $A \uplus B$ corresponds to the disjunction $A \vee B$.
- The unit type $\top$ corresponds to the trivial proposition *true*.
- The empty type $\bot$ corresponds to the absurd proposition *false*.

In particular, the type $A \to \bot$ corresponds to the negation $\neg A$.

This remarkable idea, now known as the propositions-as-types principle, the proofs-as-programs principle, or the Curry-Howard(-de Bruijn) correspondence, allows us to use a type checker also as a proof checker: a correct proof corresponds to a well-typed program, so we can use the typechecker to check whether any given proof is correct.

It is essential that all functions are total for the propositions-as-types principle to be valid: a function that's only partially defined or that doesn't terminate isn't worth anything as a proof. So by restricting our language to total functions, we have gained the ability to use well-typed functions as valid mathematical proofs.

There is a third component to the Curry-Howard correspondence besides propositions-as-types and proofs-as-programs: evaluation of a program corresponds to a simplification of the proof. This means that evaluation can turn an indirect proof into a direct one. For example, if we have proofs $a : A$ and $b : B$, then we can prove A by first constructing the pair $(a, b) : A \times B$ and then applying the function $\texttt{fst} : A \times B \to A$ to it. But of course there is a simpler proof of A: a itself. So by evaluating $\texttt{fst}\ (a, b)$ to a, we have turned an indirect proof into a direct one.

When viewed through the lens of propositions-as-types, STLC is an intuitionistic logic in the sense of Brouwer (1923). This means certain laws such as the excluded middle (for all A, we have $A \vee \neg A$) fail to hold. The reason is the decidedly constructive character of our logic: a proof of $A \vee B$ (i.e. an element of type $A \uplus B$) can be evaluated to get either an element of type A or of type B. In particular, an element of type $A \uplus (A \to \bot)$ gives us an effective way

to decide whether the type A is inhabited. Hence the removal of the excluded middle is not as a limitation of the theory, but rather a broadening of the scope: we can talk about arbitrary types, not just the ones with decidable equality (i.e. the ones for which the excluded middle holds)!

The propositions-as-types principle runs both much deeper and much broader than the link between simply typed lambda calculus and intuitionistic propositional logic. Since its discovery, many other links between logics on the one hand and type systems on the other have been discovered. Wadler (2015) gives an excellent survey of the subject. For the purpose of this thesis in particular, we are interested in one powerful extension: the correspondence between predicate logic and dependent types.

1.1.3 Dependent types

If we look at predicate logic (i.e. the kind of logic with 'for all' and 'there exists' quantifiers) through the lens of propositions-as-types, we get *dependent types*. These dependent types were first discovered by Howard (1969) and de Bruijn (1970) and developed into a full system by Martin-Löf (1972), now known as Martin-Löf type theory (MLTT).

As the name suggests, dependent types are types that *depend* on some value. The classic example of a dependent type is the type `Vec` $A\ n$ of vectors consisting of exactly n elements of type A, where n is an arbitrary expression denoting a natural number such as 3 or $1 + 2 * k$.

Dependent function types. The dependent function type $(x : A) \to B\ x$ is a generalization of the regular function type where the result type $B\ x$ can depend on the argument x. A function $f : (x : A) \to B\ x$ can be applied to an element $a : A$ to get an element $f\ a$ of type $B\ a$. For example, $(n : \mathbb{N}) \to A \to$ `Vec` $A\ n$ could be the type of a function `replicate` that returns a vector consisting of n copies of the same element of A.

Under propositions-as-types, a dependent function type corresponds to a universal quantification: a function of type $(x : A) \to B\ x$ corresponds to a proof that for all $x : A$, the property $B\ x$ holds. For example, let $m \leq n$ be a type that expresses the usual order on natural numbers $m, n : \mathbb{N}$, then a function of type $(n : \mathbb{N}) \to$ `zero` $\leq n$ is a proof that `zero` is less than or equal to any natural number.

If a function has multiple arguments of the same type, we write its type as $(x\ y : A) \to B\ x\ y$ instead of $(x : A) \to (y : A) \to B\ x\ y$. For example, a

function of type $(m\ n : \mathbb{N}) \to m \leq n \to \texttt{suc}\ m \leq \texttt{suc}\ n$ is a proof that for any m and n such that $m \leq n$, we have $\texttt{suc}\ m \leq \texttt{suc}\ n$ as well.

Dependent pair types. The dependent pair type $\Sigma_{x:A}\ (B\ x)$ consists of pairs (a, b) where $a : A$ and $b : B\ a$. Under propositions-as-types, this corresponds to an existential quantification: an element of type $\Sigma_{x:A}\ (B\ x)$ is a proof that there exists some $x : A$ such that $B\ x$ holds.

Indexed datatypes. Inductive families of datatypes (or indexed datatypes for short) are families of dependent types inductively defined by a number of constructors (Dybjer, 1991). For example, $\mathbb{N}$ is a datatype with two constructors `zero` and `suc`:

$$\begin{aligned}&\textbf{data}\ \mathbb{N} : \texttt{Set}\ \textbf{where}\\ &\quad \texttt{zero} \ : \mathbb{N}\\ &\quad \texttt{suc}\ \ : \mathbb{N} \to \mathbb{N}\end{aligned} \tag{1.2}$$

As the name suggests, indexed datatypes can also have indices. For example, $\texttt{Vec}\ A\ n$ is a datatype with one index $n : \mathbb{N}$ and two constructors `nil` and `cons`:

$$\begin{aligned}&\textbf{data}\ \texttt{Vec}\ A : \mathbb{N} \to \texttt{Set}\ \textbf{where}\\ &\quad \texttt{nil}\ \ : \texttt{Vec}\ A\ \texttt{zero}\\ &\quad \texttt{cons}\ : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)\end{aligned} \tag{1.3}$$

In contrast to n, the type A a *parameter* rather than an index. In particular, this means the type A is the same for all constructors, while the value of the index n can depend on the choice of constructor and its arguments.

Because of the propositions-as-types principle, we can also define new predicates as indexed datatypes. For example, we define the datatype $m \leq n$ with two indices $m, n : \mathbb{N}$ by the two constructors `lz` and `ls`:

$$\begin{aligned}&\textbf{data}\ \_ \leq \_ : \mathbb{N} \to \mathbb{N} \to \texttt{Set}\ \textbf{where}\\ &\quad \texttt{lz}\ : (n : \mathbb{N}) \to \texttt{zero} \leq n\\ &\quad \texttt{ls}\ : (m\ n : \mathbb{N}) \to m \leq n \to \texttt{suc}\ m \leq \texttt{suc}\ n\end{aligned} \tag{1.4}$$

The constructors `lz` and `ls` are the two basic proof principles we can use to prove inequalities; the fact that $\leq$ is an inductive family means that it is the *least* predicate on natural numbers satisfying these two principles.

Universes. An important feature of type theory is the ability to define functions that manipulate and return types. For example, $\lambda A.\ \texttt{Vec}\ A\ 3$ is a function that takes as argument a type A and returns the type $\texttt{Vec}\ A\ 3$. To give a type to such functions working on types, we use a *universe* `Set`, i.e. a type whose

elements are themselves types. In particular, `Set` consists of all base types `Bool`, $\mathbb{N}$, ..., as well as $(x : A) \to B\ x$ and $\Sigma_{x:A}\ (B\ x)$ whenever A and $B\ x$ are themselves elements of `Set`.

Although `Set` is itself a type, it is inconsistent to allow `Set` to have type `Set` itself (Girard, 1972; Coquand, 1986). Instead we can have a hierarchy of universes $\texttt{Set}_0 : \texttt{Set}_1$, $\texttt{Set}_1 : \texttt{Set}_2$, $\texttt{Set}_2 : \texttt{Set}_3$, ... Most of the time, one universe is sufficient so we use $\texttt{Set} = \texttt{Set}_0$.

Definitional and propositional equality. In (intentional) type theory, there are two distinct notions of equality. On the one hand, two terms s and t are *definitionally equal* (or *convertible*) if they both compute to the same term. For example, $(\lambda x.\ \texttt{suc zero})\ \texttt{zero}$ and $(\lambda x.\ \texttt{suc}\ x)\ \texttt{zero}$ are definitionally equal because both compute to `suc zero`. We reserve the equality sign $=$ for definitional equality. Since definitional equality can be checked automatically by the typechecker, definitionally equal terms can be used interchangeably in most contexts.

On the other hand, the *identity type* $x \equiv_A y$ expresses the property that x and y are equal elements of type A (Martin-Löf, 1984). If x and y are definitionally equal, it has a term $\texttt{refl} : x \equiv_A y$ (short for reflexivity); for example `refl` is a proof of $\texttt{zero} \equiv_{\mathbb{N}} \texttt{zero}$. On the other hand, if x and y are provably unequal then $x \equiv_A y$ is an empty type; for example $\texttt{zero} \equiv_{\mathbb{N}} \texttt{suc zero}$ is a type with no elements. If we have a term of type $u \equiv_A v$, then we call u and v *propositionally equal*.

The identity type comes equipped with a number of useful reasoning principles:

- $\texttt{refl} : \{x : A\} \to x \equiv_A x$ expresses the reflexivity of propositional equality.[2]
- $\texttt{sym} : \{x\ y : A\} \to x \equiv_A y \to y \equiv_A x$ expresses the symmetry of propositional equality.
- $\texttt{trans} : \{x\ y\ z : A\} \to x \equiv_A y \to y \equiv_A z \to x \equiv_A z$ expresses the transitivity of propositional equality.
- $\texttt{cong} : (f : A \to B) \to \{x\ y : A\} \to x \equiv_A y \to f\ x \equiv_B f\ y$ expresses congruence: applying a function to equal arguments gives equal results.

[2] As in Agda, we use curly brackets {} to indicate implicit arguments. The values of these arguments can always be deduced from the types of the other arguments, so we omit them when applying the function. For example, we write $\texttt{sym}\ e$ instead of $\texttt{sym}\ x\ y\ e$. In case we do want to make these arguments explicit, we write them between curly brackets as well, for example $\texttt{sym}\ \{x\}\ \{y\}\ e$.

- `subst` : $(P : A \to \mathtt{Set}_\ell) \to \{x\ y : A\} \to x \equiv_A y \to P\ x \to P\ y$ expresses substitution: if two types are equal up to some propositionally equal terms, then we can transport elements from one type to the other.
- `coerce` : $\{X\ Y : \mathtt{Set}_\ell\} \to (X \equiv_{\mathtt{Set}} Y) \to X \to Y$ allows us to coerce a term from one type to another if the types are propositionally equal.

Two definitionally equal terms are always propositionally equal, as expressed by `refl`. On the other hand, two terms can be propositionally equal without being definitionally equal. For example, it is possible to construct a term of type $m + n \equiv_{\mathbb{N}} n + m$ for arbitrary $m, n : \mathbb{N}$, proving that $+$ on natural numbers is commutative. However, if m and n are variables then $m + n$ and $n + m$ will never evaluate to the same result no matter how hard we try, so they are not definitionally equal. For the types we have seen so far, the two notions of equality coincide on closed terms (i.e. ones that don't contain any free variables). But even this ceases to be true once we move on to homotopy type theory (Section 1.3).

1.2 Pattern matching and unification

> *That is the very purpose of declarative programming — to make it more likely that we mean what we say by improving our ability to say what we mean.*
>
> — Conor McBride (2003)

Dependent type theory as presented in the previous section is a powerful type system for both programs and proofs. But how do we actually write these programs and proofs? So far, we have only seen a few basic constructions such as function application, lambda abstraction, and basic constructors such as `true`, `false`, `zero`, `suc`, ... But real programs do more than construct values and pass them around: they can analyse values and recurse on them, producing new values based on the results. Similarly, to write proofs we need some way to do case analysis and induction on a given value.

In Martin-Löf type theory, these programs and proofs are written by using the elimination principles for each type, or *eliminators* for short. These eliminators are powerful enough to write any program or proof we want, but they are difficult to use and to read. In effect, these eliminators are the assembly language of type theory.

To make it easier to write dependently typed programs, people have developed higher-level techniques that can be translated to eliminators behind the scenes. In this thesis, we study two of these techniques in detail: dependent pattern matching and specialization by unification. There are other possible techniques besides these two: Coq for example makes heavy use of *tactics*, scripts that generate low-level type-theoretic code.

1.2.1 Pattern matching = case analysis + recursion

Pattern matching is the basic way to write functions on algebraic datatypes in functional programming languages such as Haskell and ML. A definition by pattern matching consists of a set of equalities called *clauses* that the function has to satisfy. It can be adapted directly to type theory to provide a convenient way to define functions on simple datatypes such as $\mathtt{Bool}$ and $\mathbb{N}$. Because pattern matching is such a central topic in this thesis, we introduce it here by plenty of examples. A general definition can be found in Chapter 2.

The most fundamental feature of pattern matching is that it allows us to define functions by case analysis.

Example 1.1. We define the function $\mathtt{not}$ by pattern matching as follows:

$$\begin{aligned} &\mathtt{not} : \mathtt{Bool} \to \mathtt{Bool} \\ &\mathtt{not}\ \mathtt{true}\ \ = \mathtt{false} \\ &\mathtt{not}\ \mathtt{false} = \mathtt{true} \end{aligned} \tag{1.5}$$

This definition consists of a *type signature* $\mathtt{not} : \mathtt{Bool} \to \mathtt{Bool}$ and the two clauses $\mathtt{not}\ \mathtt{true} = \mathtt{false}$ and $\mathtt{not}\ \mathtt{false} = \mathtt{true}$. The left-hand side of each clause consists of the function being defined applied to some arguments called the *patterns*. For example, the patterns of $\mathtt{not}$ are $\mathtt{true}$ and $\mathtt{false}$.

Aside from constructors, patterns can also contain *pattern variables*.

Example 1.2. We define the function $\mathtt{is\text{-}zero}$ on natural numbers by pattern matching:

$$\begin{aligned} &\mathtt{is\text{-}zero} : \mathbb{N} \to \mathtt{Bool} \\ &\mathtt{is\text{-}zero}\ \mathtt{zero}\ \ \ \ = \mathtt{true} \\ &\mathtt{is\text{-}zero}\ (\mathtt{suc}\ m) = \mathtt{false} \end{aligned} \tag{1.6}$$

The second clause defines $\mathtt{is\text{-}zero}$ for any number of the form $\mathtt{suc}\ m$ such as $\mathtt{suc}\ \mathtt{zero}$, $\mathtt{suc}\ (\mathtt{suc}\ \mathtt{zero})$, ...

Patterns are required to consist of constructors and variables only, and each variable may occur only once in a pattern. This is to avoid having to compare

arbitrary terms when evaluating a function by pattern matching. Additionally, since all functions are required to be total there has to be a clause for each constructor.

There can also be recursive calls to the function being defined on the right-hand side of a clause.

Example 1.3. We define the function `half` on natural numbers by recursion:

$$\begin{aligned} &\texttt{half} : \mathbb{N} \to \mathbb{N} \\ &\texttt{half zero} &&= \texttt{zero} \\ &\texttt{half (suc zero)} &&= \texttt{zero} \\ &\texttt{half (suc (suc } n\texttt{))} &&= \texttt{suc (half } n\texttt{)} \end{aligned} \tag{1.7}$$

This example also shows that we can make further case distinctions by using nested patterns such as `suc zero` and `suc (suc` m`)`.

Functions defined by pattern matching may have multiple arguments, and we can do case analysis on any of them.

Example 1.4. We define addition on natural numbers as follows:

$$\begin{aligned} &\_ + \_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &\texttt{zero} + n &&= n \\ &(\texttt{suc}\ m) + n &&= \texttt{suc}\ (m + n) \end{aligned} \tag{1.8}$$

We borrow the notation for infix functions from the Agda language: $\_ + \_$ stands for the function adding together two arguments; the underscores tell us where the arguments go. More precisely, $m + n$ is equivalent to $(\_ + \_)\ m\ n$.

If the return type of a function is dependent on one of the arguments, then the type of the right-hand sides of the clauses can be different for each clause.

Example 1.5. We define the function `replicate` as follows:

$$\begin{aligned} &\texttt{replicate} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \\ &\texttt{replicate zero}\ x &&= \texttt{nil} \\ &\texttt{replicate (suc}\ m\texttt{)}\ x &&= \texttt{cons}\ m\ x\ \texttt{(replicate}\ m\ x\texttt{)} \end{aligned} \tag{1.9}$$

The type of the first clause is `Vec` A `zero`, and the type of the second clause is `Vec` A `(suc` m`)`. This would be impossible in a language without dependent types.

Under propositions-as-types, a function corresponds to a proof of an implication, so we can use pattern matching to write proofs as well as programs.

Example 1.6. We prove a lemma (named `not-not` here) that negating a boolean twice always gives back the same boolean we started with:

$$\begin{aligned}&\texttt{not-not} : (b : \texttt{Bool}) \to \texttt{not}\ (\texttt{not}\ b) \equiv_{\texttt{Bool}} b\\&\texttt{not-not}\ \texttt{true}\ \ = \texttt{refl}\\&\texttt{not-not}\ \texttt{false} = \texttt{refl}\end{aligned} \tag{1.10}$$

The type of the first `refl` is `not (not true)` $\equiv_{\texttt{Bool}}$ `true`, and similarly the type of the second `refl` is `not (not false)` $\equiv_{\texttt{Bool}}$ `true`. The constructor `refl` : $x \equiv_A x$ has the correct type to be used here because `not (not true)` is *definitionally* equal to `true`. If they were only propositionally equal, then we wouldn't be able to write `refl` here.

For the empty type $\bot$ there are no constructors, so we can do a case analysis on it using zero cases. Instead of writing no clauses at all, we use the *absurd pattern* () to indicate that there are no constructors.

Example 1.7. We define a function `absurd` that expresses the principle of 'ex falso quodlibet': from a false assumption we can derive anything.

$$\begin{aligned}&\texttt{absurd} : (A : \texttt{Set}_\ell) \to \bot \to A\\&\texttt{absurd}\ A\ ()\end{aligned} \tag{1.11}$$

We can write proofs by induction as recursive functions.

Example 1.8. From the definition of $+$, it follows that $\texttt{zero} + n = n$, but we don't get that $n + \texttt{zero} = n$. Using pattern matching, we prove that $n + \texttt{zero} \equiv_{\mathbb{N}} n$ for any $n : \mathbb{N}$:

$$\begin{aligned}&\texttt{plus-zero} : (n : \mathbb{N}) \to n + \texttt{zero} \equiv_{\mathbb{N}} n\\&\texttt{plus-zero}\ \texttt{zero}\ \ \ \ = \texttt{refl}\\&\texttt{plus-zero}\ (\texttt{suc}\ m)\ = \texttt{cong}\ \texttt{suc}\ (\texttt{plus-zero}\ m)\end{aligned} \tag{1.12}$$

The recursive call `plus-zero` m has type $m + \texttt{zero} \equiv_{\mathbb{N}} m$, which can be transformed into an element of type $\texttt{suc}\ m + \texttt{zero} \equiv_{\mathbb{N}} \texttt{suc}\ m$ by applying `cong suc` to it. The argument m of the recursive call is strictly smaller than the pattern `suc` m on the left-hand side. Were this not the case, then the recursion would go on forever and the proof would be invalid.

1.2.2 Matching on indexed datatypes

Things get more complicated when we define functions by pattern matching on an indexed datatype such as `Vec` $A\ n$ or $m \leq n$. This is called *dependent pattern matching* (Coquand, 1992).

Example 1.9. As a first example, we define the concatenation of two vectors as follows:

$$\begin{aligned}&\texttt{concat} : (m\ n : \mathbb{N})(xs : \texttt{Vec}\ A\ m)(ys : \texttt{Vec}\ A\ n) \to \texttt{Vec}\ A\ (m + n)\\ &\texttt{concat}\ .\texttt{zero}\ n\ \texttt{nil}\ ys \quad = \ ys\\ &\texttt{concat}\ .(\texttt{suc}\ m)\ n\ (\texttt{cons}\ m\ x\ xs)\ ys \ = \ \texttt{cons}\ (m + n)\ x\ (\texttt{concat}\ xs\ ys)\end{aligned} \tag{1.13}$$

In the first clause, the argument $m : \mathbb{N}$ is required to be equal to `zero` for the pattern `nil` to be well-typed. This is indicated by the dot in the *inaccessible pattern* `.zero`. Similarly, the inaccessible pattern $.(\texttt{suc}\ m)$ in the second clause witnesses the fact that $\texttt{suc}\ m$ is the only type-correct argument that can occur in that position. The variable m occurs twice on the left-hand side in the clause for `cons`. This is ordinarily not allowed, but it is fine here because one occurrence is inside an inaccessible pattern.

When evaluating the function `concat`, the arguments corresponding to these inaccessible patterns can be safely ignored because the type system guarantees that they must have the specified value. For example, whenever we have a well-typed application of the form $\texttt{concat}\ m\ n\ \texttt{nil}\ ys$, the argument m is guaranteed to be equal to `zero`.

Dependent types allow us to work around the limitations of a total language by restricting the type of the input in such a way that the function becomes total. For example, it is possible that there is no way that a particular constructor can be used in a well-typed manner. In that case, we skip the corresponding clause.

Example 1.10. We define a tail function on vectors as follows:

$$\begin{aligned}&\texttt{tail} : (n : \mathbb{N}) \to \texttt{Vec}\ A\ (\texttt{suc}\ n) \to \texttt{Vec}\ A\ n\\ &\texttt{tail}\ .m\ (\texttt{cons}\ m\ x\ xs) \ = \ xs\end{aligned} \tag{1.14}$$

It is allowed to skip the clause for $\texttt{nil} : \texttt{Vec}\ A\ \texttt{zero}$ because there is no way that $\texttt{suc}\ n$ can be equal to `zero`. This is all the better because there is no way to take the tail of an empty vector!

In some cases it is even possible that all cases can be skipped. In such cases we again use the absurd pattern ().

Example 1.11. We prove that a natural number of the form $\texttt{suc}\ n$ can never be less than or equal to `zero`:

$$\begin{aligned}&\texttt{not-less} : (n : \mathbb{N}) \to \texttt{suc}\ n \leq \texttt{zero} \to \bot\\ &\texttt{not-less}\ n\ ()\end{aligned} \tag{1.15}$$

Neither `lz` nor `ls` can be used in a well-typed manner at type $\texttt{suc}\ n \leq \texttt{zero}$, so both cases can be skipped.

By first doing a case analysis on one argument, we can learn something about the type of another argument. This information can then be used to exclude some of the cases for that second argument.

Example 1.12. We prove antisymmetry of the relation $\leq$ on natural numbers as follows:

$$\begin{aligned}&\texttt{antisym} : (m\ n : \mathbb{N}) \to m \leq n \to n \leq m \to m \equiv_{\mathbb{N}} n\\&\texttt{antisym}\ \texttt{.zero}\ \texttt{.zero}\ (\texttt{lz}\ \texttt{.zero})\ (\texttt{lz}\ \texttt{.zero}) = \texttt{refl}\\&\texttt{antisym}\ \texttt{.(suc}\ k\texttt{)}\ \texttt{.(suc}\ l\texttt{)}\ (\texttt{ls}\ k\ l\ x)\ (\texttt{ls}\ \texttt{.}l\ \texttt{.}k\ y) =\\&\qquad \texttt{cong suc}\ (\texttt{antisym}\ k\ l\ x\ y)\end{aligned} \tag{1.16}$$

In the first clause, matching with the constructor `lz` on the argument of type $m \leq n$ teaches us that m is equal to zero (this is expressed by the first inaccessible pattern `.zero`). By instantiating m to be `zero`, the type $n \leq m$ of the final argument becomes $n \leq \texttt{zero}$, so we exclude the case for the constructor $\texttt{ls} : (m\ n : \mathbb{N}) \to m \leq n \to \texttt{suc}\ m \leq \texttt{suc}\ n$. Similarly, in the second clause we exclude the case for the constructor `lz` because m and n have been instantiated to $\texttt{suc}\ k$ and $\texttt{suc}\ l$ respectively.

The identity type $x \equiv_A y$ can be seen as an indexed datatype with one constructor `refl`. This allows us to write proofs about propositional equality by pattern matching on this constructor. Matching on an argument of type $x \equiv_A y$ with `refl` causes x to be unified with y, forcing them to be equal.

Example 1.13. We define the functions `sym`, `trans`, `cong`, and `subst` by pattern matching on `refl`:

$$\begin{aligned}&\texttt{sym} : \{x\ y : A\} \to x \equiv_A y \to y \equiv_A x\\&\texttt{sym refl} = \texttt{refl}\end{aligned} \tag{1.17}$$

$$\begin{aligned}&\texttt{trans} : \{x\ y\ z : A\} \to x \equiv_A y \to y \equiv_A z \to x \equiv_A z\\&\texttt{trans refl refl} = \texttt{refl}\end{aligned} \tag{1.18}$$

$$\begin{aligned}&\texttt{cong} : (f : A \to B)\{x\ y : A\} \to x \equiv_A y \to f\ x \equiv_B f\ y\\&\texttt{cong}\ f\ \texttt{refl} = \texttt{refl}\end{aligned} \tag{1.19}$$

$$\begin{aligned}&\texttt{subst} : (P : A \to \texttt{Set})\{x\ y : A\} \to x \equiv_A y \to P\ x \to P\ y\\&\texttt{subst}\ P\ \texttt{refl}\ p = p\end{aligned} \tag{1.20}$$

$$\begin{aligned}&\texttt{coerce} : \{X\ Y : \texttt{Set}\} \to (X \equiv_{\texttt{Set}} Y) \to X \to Y\\&\texttt{coerce refl}\ x = x\end{aligned} \tag{1.21}$$

Interpreting the identity as an inductive type with the single constructor `refl` has implications on what we can prove about it, and by extension on the nature of the type theory as a whole. If we're not careful, these implications conflict with other interpretations of the identity type, such as the one given by *homotopy type theory* (see next section).

1.2.3 Specialization by unification

When splitting on a type from an inductive family, the typechecker needs to determine which constructors can occur in a given position and how the variables need to be instantiated for the pattern to be well-typed. To do this, it applies *unification* to the indices of the datatype in question. Unification is the process of searching for *unifiers*, i.e. substitutions that make the left- and right-hand side of an equation (definitionally) equal. If unification determines that there can be no such substitution, then we can skip the case for the corresponding constructor. This method of solving equations to either gain more information about the type of the right-hand side or to derive an absurdity is called *specialization by unification* (Goguen, McBride, and McKinna, 2006).

Example 1.14. In the definition of `tail` (Example 1.10), the typechecker has to determine whether there should be a case for the constructor `nil` : `Vec` A `zero` for the argument of type `Vec` A (`suc` n). To do this, it tries to unify `zero` with `suc` n. In this case unification detects an absurdity because `zero` can never be equal to `suc` n; this is called the **conflict** rule. On the other hand, for the constructor `cons` : $(m : \mathbb{N})(x : A)(xs :$ `Vec` $A\ m) \to$ `Vec` A (`suc` m), unification of `suc` m with `suc` n succeeds by substituting m for n; this is called the **injectivity** rule.

Example 1.15. In the definition of `antisym` (Example 1.12), it is allowed to skip the two cases where one of the arguments is `lz` n and the other is `ls` $l\ k$ because `zero` can never be of the form `suc` k (by the **conflict** rule). In the second clause, the first argument of the second `ls` is replaced by $.l$ because of the **injectivity** rule, and similarly its second argument is replaced by $.k$.

In this thesis, we present two different unification algorithms that can be used to check definitions by dependent pattern matching. The first one is a simple untyped unification algorithm that is very similar to the one used by McBride (2000), but it has three additional restrictions that make it useable in any version of Martin-Löf type theory (Section 2.2.3 and Section 2.3). The second one is a more powerful *typed* unification algorithm and is the main subject of Chapter 3. It includes all the rules of the first algorithm and adds more general rules for indexed datatypes and record types.

1.3 Homotopy type theory

> *Homotopy type theory is a programming language for which we don't yet know how to run the programs. Most mathematicians are quite blasé about this, but computer scientists feel somehow irritated by this fact.*
>
> — Thorsten Altenkirch

From what we've seen so far, the nature of the identity type $x \equiv_A y$ is still somewhat mysterious: we know some of its properties such as reflexivity, symmetry, transitivity, congruence, and substitutivity, but we don't know much about its structure like we do about a type like $A \uplus B$. For example, what can we tell about the type $f \equiv_{A \to B} g$ when the functions f and g are pointwise equal but not definitionally so? And what about the type $A \equiv_{\mathtt{Set}} B$ when A and B are isomorphic yet unequal types? One possible answer to these questions (and many more) is given by an emerging field called *homotopy type theory* (HoTT) (The Univalent Foundations Program, 2013).

Homotopy type theory is based on a new interpretation of types, different from the two we have already seen. Instead of viewing types as *sets of values* or as *propositions*, this interpretation views types as *topological spaces* (such as an interval, a circle, or a torus). Under this interpretation, a term $u : A$ corresponds to a point in the topological space A, and an element of type $u \equiv_A v$ corresponds to a *continuous path* in A from u to v. Any function $f : A \to B$ maps not only points in A to points in B, but also paths of type $u \equiv_A v$ to paths of type $f\ u \equiv_B f\ v$ by `cong` (1.19), so any well-typed function represents a (path)-*continuous map* between topological spaces.

1.3.1 The univalence axiom

One of the core elements of HoTT is the *univalence axiom* proposed by Vladimir Voevodsky. It gives a surprising yet natural interpretation of the identity type $A \equiv_{\mathtt{Set}} B$ as the type of *equivalences* between the two types A and B. To state it precisely, we first need to define what we mean by an equivalence.

In mathematics, an equivalence is usually defined as a function $f : A \to B$ that has an inverse $g : B \to A$, meaning that $g\ (f\ x) = x$ for all $x : A$ and $f\ (g\ y) = y$ for all $y : B$. For technical reasons, homotopy type theory uses a definition that is at first sight a bit more liberal (The Univalent Foundations Program, 2013, Definition 4.3.1):

Definition 1.16 (Equivalence)**.** A function $f : A \to B$ is an *equivalence* if we have two functions $g_1 : B \to A$ and $g_2 : B \to A$ that are respectively a left and a right inverse of f (i.e. we have terms of type $(x : A) \to g_1\ (f\ x) \equiv_A x$ and $(y : B) \to f\ (g_2\ y) \equiv_B y$). Two types A and B are *equivalent* if there exists an equivalence from A to B. The type of all equivalences $f : A \to B$ is written as $A \simeq B$.

If $f : A \simeq B$ is an equivalence, then we also write f for the function $f : A \to B$. We write $\texttt{linv}\ f$ for the function g_1 (for the **l**eft **inv**erse of f), and $\texttt{isLinv}\ f$ for the proof of $(x : A) \to \texttt{linv}\ f\ (f\ x) \equiv_A x$. Similarly, $\texttt{rinv}\ f$ stands for the function g_2 and $\texttt{isRinv}\ f$ for the proof of $(y : B) \to f\ (\texttt{rinv}\ f\ y) \equiv_B y$.

While the left and right inverse of an equivalence are allowed to be different, their results are always propositionally equal.

Lemma 1.17. *If $f : A \simeq B$ is an equivalence, then we can construct a proof of $(y : B) \to \texttt{linv}\ f\ y \equiv_A \texttt{rinv}\ f\ y$.*

Construction. Let $y : B$. By $\texttt{isRinv}\ f\ y$, we have $f\ (\texttt{rinv}\ f\ y) \equiv_B y$, hence by $\texttt{cong}$, we have $\texttt{linv}\ f\ (f\ (\texttt{rinv}\ f\ y)) \equiv_A \texttt{linv}\ f\ y$. By $\texttt{isLinv}\ f$, we also have $\texttt{linv}\ f\ (f\ (\texttt{rinv}\ f\ y)) \equiv_A \texttt{rinv}\ f\ y$. Putting these proofs together with $\texttt{sym}$ and $\texttt{trans}$, we get that $\texttt{linv}\ f\ y \equiv_A \texttt{rinv}\ f\ y$, as we wanted to prove. □

For many equivalences, $\texttt{linv}\ f$ and $\texttt{rinv}\ f$ are actually definitionally equal. In that case we write f^{-1} for their common value.

Formally, the univalence axiom states that equality between types is equivalent with equivalence, i.e. for any two types A and B we have an equivalence:

$$\texttt{ua} : (A \simeq B) \simeq (A \equiv_{\texttt{Set}} B) \tag{1.22}$$

Moreover, for any concrete equivalence $f : A \simeq B$, coercing by $\texttt{ua}\ f$ is the same as applying f directly:

$$\texttt{coerce}\ (\texttt{ua}\ f)\ x = f\ x \tag{1.23}$$

Univalence captures the common mathematical practice of informal reasoning "up to isomorphism" in a nice and formal way. It also has a number of useful consequences, such as *functional extensionality*:

Theorem 1.18 (Functional extensionality)**.** *Assume the univalence axiom (1.22). For any two functions $f, g : A \to B$ we have an equivalence:*

$$\texttt{funext} : ((x : A) \to f\ x \equiv_B g\ x) \simeq (f \equiv_{A \to B} g) \tag{1.24}$$

Proof. See The Univalent Foundations Program (2013, Section 4.9). □

However, from the point of view of type theory as a programming language there is a big problem with the univalence axiom: the fact that it is an axiom. This means that it is a term without any computational behaviour, i.e. a program for which we don't know how to run it. For several years, it remained an open problem how to give a computational interpretation to univalence, until it was finally solved by Cohen, Coquand, Huber, and Mörtberg (2016) in the form of *cubical type theory*. This is a new version of Martin-Löf type theory with additional primitive constructions that allow us to prove univalence, thus turning it from an axiom into a theorem. So cubical type theory allows us to reap the benefits of homotopy type theory without losing the computational nature of the older dependent type theories.

In this thesis, we use concepts from homotopy type theory such as equivalence and equalities "lying over" another equality. The notation for telescopic equality used in Chapter 3 is also inspired by cubical type theory. However, our work doesn't require any primitives on top of basic intuitionistic type theory (such as univalence). In fact, our work can be equally well understood without any knowledge of HoTT, and is still useful in a setting that assumes entirely different axioms (such as the law of the excluded middle from classical logic).

1.3.2 Pattern matching in HoTT

Many basic constructions in HoTT can be written elegantly using pattern matching, for example `sym` (1.17) corresponds to the reversal of a path, `trans` (1.18) corresponds to the composition of two paths, and `cong` (1.19) is a proof that all functions in HoTT are (path-)continuous.

However, the original version of dependent pattern matching by Coquand (1992) is *incompatible* with the univalence axiom. Consider an arbitrary type A and a path $e : A \equiv_{\mathtt{Set}} A$ from A to itself. By pattern matching on `refl`, we prove that coercing along this path is the identity function:

$$\begin{aligned} &\mathtt{coerce\text{-}id} : (e : A \equiv_{\mathtt{Set}} A)(x : A) \to \mathtt{coerce}\ e\ x \equiv_A x \\ &\mathtt{coerce\text{-}id}\ \mathtt{refl}\ x\ =\ \mathtt{refl} \end{aligned} \tag{1.25}$$

On the other hand, the function `not` : `Bool` → `Bool` (Example 1.1) is both its own left and right inverse (Example 1.6), so it is an equivalence. By univalence, we get an equality proof `ua not` : `Bool` $\equiv_{\mathtt{Set}}$ `Bool` such that `coerce` (`ua not`) `true` = `false` and vice versa. But this is a contradiction with `coerce-id`, which says that `coerce` (`ua not`) `true` $\equiv_{\mathtt{Bool}}$ `true`! So having both dependent pattern matching and univalence together leads to an unsound system.

This incompatibility has forced people working in homotopy type theory to avoid using pattern matching, instead writing their programs and proofs using the cumbersome eliminators. One of the main contributions of this thesis is a new version of dependent pattern matching that is safe to use together with univalence (Section 2.3).

1.4 Desugaring pattern matching

> *Any sufficiently advanced technology is indistinguishable from magic.*
>
> — Arthur C. Clarke (1962)

Designers of a dependently typed language face a difficult dilemma: on the one hand, programs and proofs should be as easy and intuitive as possible to write. On the other hand, they should satisfy strict requirements, lest they violate important properties of type theory such as logical consistency and decidable type checking. This dilemma manifests itself when we compare dependent pattern matching to the standard eliminators. On the one hand, proofs by dependent pattern matching are typically much shorter and more readable than ones that use eliminators. On the other hand, dependent pattern matching as given by Coquand is incompatible with other useful assumptions such as univalence.

Example 1.19. As an example of the difficulty of using eliminators compared to pattern matching, Figure 1.1 gives an alternative definition of `antisym` (Example 1.12) that only uses eliminators. In this definition, $\texttt{elim}_{\leq}$ is the standard eliminator for the datatype $\leq$ (Definition 4.2) and $\texttt{noConf}_{\mathbb{N}}$ is the "no confusion" property for the $\mathbb{N}$ type (Lemma 4.16). This "no confusion" property can be constructed from eliminators as well, but expanding this construction here would make the proof even longer.

This proof by eliminators is more complex than the proof by pattern matching in Example 1.12. All the equational reasoning that was done automatically in the definition by pattern matching now has to be done explicitly. The proof by eliminators also requires considerable work for the construction of the first argument of each eliminator (called the *motive*), while this can be done automatically in many cases (McBride, 2002). So it is clearly preferable to use pattern matching for this proof.

A way to go around this dilemma is to provide high-level features to the user, but guarantee that any definition can be translated to an equivalent one that

$$
\begin{aligned}
&\texttt{antisym} : (m\ n : \mathbb{N}) \to m \leq n \to n \leq m \to m \equiv_{\mathbb{N}} n \\
&\texttt{antisym} = \texttt{elim}_{\leq}\ (\lambda m; n; \_ .\ n \leq m \to m \equiv_{\mathbb{N}} n) \\
&\quad (\lambda n; e.\ \texttt{elim}_{\leq}\ (\lambda n; m; \_ .\ m \equiv_{\mathbb{N}} \texttt{zero} \to m \equiv_{\mathbb{N}} n) \\
&\qquad (\lambda n; e.\ e) \\
&\qquad (\lambda k; l; \_; \_; e.\ \texttt{absurd}\ (\texttt{suc}\ l \equiv_{\mathbb{N}} \texttt{suc}\ k)\ (\texttt{noConf}_{\mathbb{N}}\ (\texttt{suc}\ l)\ \texttt{zero}\ e)) \\
&\qquad n\ \texttt{zero}\ e\ \texttt{refl}) \\
&\quad (\lambda m; n; \_; H; q.\ \texttt{cong}\ \texttt{suc}\ (H \\
&\qquad (\texttt{elim}_{\leq}\ (\lambda k; l; \_ .\ k \equiv_{\mathbb{N}} \texttt{suc}\ n \to l \equiv_{\mathbb{N}} \texttt{suc}\ m \to n \leq m) \\
&\qquad\quad (\lambda \_; e; \_ .\ \texttt{absurd}\ (n \leq m)\ (\texttt{noConf}_{\mathbb{N}}\ \texttt{zero}\ (\texttt{suc}\ n)\ e)) \\
&\qquad\quad (\lambda k; l; e; \_; x; y.\ \texttt{subst}\ (\lambda n.\ n \leq m) \\
&\qquad\qquad (\texttt{noConf}_{\mathbb{N}}\ (\texttt{suc}\ k)\ (\texttt{suc}\ n)\ x) \\
&\qquad\qquad (\texttt{subst}\ (\lambda m.\ k \leq m)\ (\texttt{noConf}_{\mathbb{N}}\ (\texttt{suc}\ l)\ (\texttt{suc}\ m)\ y)\ e)) \\
&\qquad\quad (\texttt{suc}\ n)\ (\texttt{suc}\ m)\ y\ \texttt{refl}\ \texttt{refl})))
\end{aligned}
\tag{1.26}
$$

Figure 1.1: This definition of the function `antisym` is more complex than the one by pattern matching (Example 1.12).

only uses simple, well-understood primitives. McBride (2000) and Goguen et al. (2006) applied this idea to dependent pattern matching: they show that any valid definition by dependent pattern matching can be translated to one that only uses eliminators. However, in this translation they depend on a non-standard eliminator for the identity type called the K rule. So if we want to understand why the original version of dependent pattern matching is incompatible with univalence, we first have to understand this K rule.

The standard eliminator for the identity type $x \equiv_A y$ is called the J rule:

$$\mathrm{J} : (P : (y : A) \to x \equiv_A y \to \texttt{Set})(p : P\ x\ \texttt{refl})(y : A)(e : x \equiv_A y) \to P\ y\ e \tag{1.27}$$

This rule is a generalization of `subst` where the type P is allowed to depend on the given equality proof. Using only J, it is possible to define `sym`, `trans`, `cong`, and `subst`, and `coerce`, but not `coerce-id`.

Where the J rule allows us to prove properties of a general equality proof of $x \equiv_A y$, the K rule tells us something more specific about proofs of reflexive equations $x \equiv_A x$:

$$\mathrm{K} : (P : x \equiv_A x \to \texttt{Set})(p : P\ \texttt{refl})(e : x \equiv_A x) \to P\ e \tag{1.28}$$

The K rule is equivalent with the *uniqueness of identity proofs* principle (UIP), which states that any two proofs of $x \equiv_A y$ are equal. This is clearly incompatible with the interpretation of types as topological spaces: any topological space

with a hole in it (such as the circle) has infinitely many distinct paths between any two given points.

Coquand (1992) already observed that it is easy to prove K by pattern matching:

$$\begin{aligned} &\mathrm{K} : (P : a \equiv_A a \to \mathtt{Set})(p : P\ \mathtt{refl})(e : a \equiv_A a) \to P\ e \\ &\mathrm{K}\ P\ p\ \mathtt{refl} \ = \ p \end{aligned} \tag{1.29}$$

On the other hand, it is impossible to define K by only using J (Hofmann and Streicher, 1994). This is a good thing: it means we might still use J if we assume axioms that are incompatible with K.

As mentioned before, translating a definition by dependent pattern matching to eliminators in general requires the K rule.

Example 1.20. To translate the definition of the function (1.25) to eliminators, we need to make use of K:

$$\mathtt{coerce\text{-}id} = \lambda e.\ \lambda x.\ \mathrm{K}\ (\lambda e.\ \mathtt{coerce}\ e\ x \equiv_{\mathtt{Bool}} x)\ \mathtt{refl}\ e \tag{1.30}$$

Without K, it would be impossible to define `coerce-id` in terms of eliminators.

In Chapter 4 we show how definitions by dependent pattern matching as described in this thesis can be translated to eliminators *without* making use of the K rule. This translation guarantees that dependent pattern matching can be used together with univalence, as well as with any other addition to type theory (that is compatible with the standard eliminators).

1.5 Three recurring themes

> *Classical mathematics [...] is lacking neither truth nor sense but only imagination.*
>
> — Charles McCarty

Before we proceed to the main body of this thesis, we want to draw your attention to three important themes that will occur again and again. These themes may be surprising to some and obvious to others, but in either case they are important considerations during the work on this thesis and drove many of the design decisions in it. The three themes are the following:

1. We consider proofs to be relevant, i.e. proofs are themselves interesting mathematical objects that can have a certain algorithmic content.

2. We aim for our work to be compatible with all axioms consistent with Martin-Löf type theory, without requiring any one of them in particular.

3. All constructions in this thesis are given not just to have the correct type but also to have the right computational behaviour.

1.5.1 Proof relevance and homotopy type theory

The propositions-as-types principle tells us that a proposition can be represented as a type, and a proof of the proposition as a term of that type. But this doesn't yet tell us anything about the structure of those proofs. Do they contain any information besides the bare fact that the proposition is true? In Coq, all types that live in the universe `Prop` (including the standard identity type) are considered computationally *irrelevant*, i.e. they can be erased when compiling the program. But in a constructive logic, it can make sense to not always erase proofs. For example, a proof of $(m\ n : \mathbb{N}) \to (m \leq n) \uplus (n \leq m)$ can be used as a procedure to decide which of two numbers is bigger, and can be used as part of a sorting algorithm. As another example, a proof that a version of type theory is strongly normalizing can be used as a normalization algorithm for that theory, which is an essential part of a typechecker. For this reason, we radically consider all proofs to be relevant, so we may make use of their algorithmic content when we have a need for it.

Homotopy type theory is a particularly nice example of proof relevance, because it considers equality proofs to be paths in a topological space. This means there can be many different equality proofs between two given terms, we may consider equality of equality proofs, etc. The crown jewel of HoTT is of course the univalence axiom, which allows us to use a proof of equality of two types as an *algorithm* for translating between these two types, and vice versa. None of these things would be possible if we didn't consider proofs to be relevant.

On the other hand, choosing proof relevance doesn't mean we cannot consider bare proofs without further structure if we want to. Homotopy type theory equips us with the tool of *propositional truncations*, which allows us to erase the computational contents from a type (The Univalent Foundations Program, 2013, Section 3.7). So proof irrelevance is just a special case of proof relevance.

Our quest for proof relevance becomes particularly apparent in Chapter 3, where we describe a proof-relevant unification algorithm. We give a proof-relevant interpretation of the concept of a most general unifier as an *equivalence* between two sets of equations. Such an equivalence tells how to instantiate each variable, and also gives evidence that this instantiation indeed defines a most general unifier. The fact that this evidence is proof relevant is important for

the translation of functions by dependent pattern matching to eliminators in Chapter 4.

A practical consequence of this proof-relevant approach is that many of the proofs in this thesis are labelled as 'Construction' instead of 'Proof'. This is the case for those proofs that are given as terms in the type theory we work in. On the other hand, metatheoretical proofs *about* the type theory are still labelled as 'Proof' as normal. This is particular the case when we prove that two terms are definitionally equal.

1.5.2 The blessing and the curse of axiomatic freedom

On its own, Martin-Löf type theory is a rather minimal language with only a small number of language constructs. As in all sound logical systems, there are many statements that it can neither prove nor refute. So it makes sense to extend it with additional principles, depending on your needs or philosophical convictions. We have already mentioned a good number of these principles:

- The law of the excluded middle states that every type is decidable.
- The principle of uniqueness of identity proofs (i.e. the K rule) states that any two proofs of the same equality are equal.
- The principle of proof irrelevance states that any two proofs of the same proposition are equal.
- The univalence axiom states that equivalence is equivalent with equality.

The blessing of axiomatic freedom given to us by MLTT is that we can freely choose which of these axioms we want to use. This means the same language with only small additions can be used for very different purposes.

However, using too many of these principles together blows up the whole theory, making it inconsistent. This is the curse of axiomatic freedom: all of the different dialects of MLTT can never be united into one language. For example, univalence is incompatible with the uniqueness of identity proofs. As another example, univalence is also incompatible with the law of the excluded middle (The Univalent Foundations Program, 2013, Corollary 3.2.7).

Designers of a dependently typed language face a difficult choice: they can either choose a particular set of principles and build them into the language, or provide only a basic type theory and leave the choice up to the user. The former choice is made by NuPRL (Kreitz, 2002), where any principle that holds in a

specific preferred model can be proven in the language. On the other hand, Coq and Agda take the latter choice and let the user decide by adding postulates to the language. This allows for experimentation with many different (even incompatible) extensions of type theory within the same framework. Because improving the pattern matching in Agda is one of our main motivations, this is also the choice we make in this thesis.

As another example of the curse of axiomatic freedom, dependent pattern matching can be made more powerful by allowing the unification algorithm used to check case splits to postpone and reorder equations. However, doing so allows one to prove a principle called *injectivity of type constructors* (Example 3.3). This principle allows us for example to deduce from $\texttt{List}\ A \equiv_{\texttt{Set}} \texttt{List}\ B$ that $A \equiv_{\texttt{Set}} B$. This principle is syntactic in nature and is incompatible with both univalence and the excluded middle:

Theorem 1.21. *MLTT extended with univalence and injective type constructors is inconsistent.*

The proofs of this theorem and the next one are not essential for the rest of this thesis. However, to our knowledge there is no easy reference for them and we think they are interesting enough to mention here.

Proof. Let $\texttt{D} : \texttt{Set} \to \texttt{Set}$ be an inductive family with no constructors. Then $\texttt{D}\ \top \simeq \bot \simeq \texttt{D}\ \bot$, so $\texttt{D}\ \top \equiv_{\texttt{Set}} \texttt{D}\ \bot$ by univalence. But if $\texttt{D}$ is injective, this means that $\top \equiv_{\texttt{Set}} \bot$, which is clearly a contradiction. □

Theorem 1.22. *MLTT extended with the excluded middle and injective type constructors is inconsistent.*

Proof. This proof is based on the proof given by Hur (2010).

Assume the datatype $\texttt{D} : (\texttt{Set} \to \texttt{Set}) \to \texttt{Set}$ is injective (it doesn't matter what the constructors of $\texttt{D}$ are). We define a right inverse $\texttt{E}$ of $\texttt{D}$ as follows: if A is equal to $\texttt{D}\ F$ for some $F : \texttt{Set} \to \texttt{Set}$, then $\texttt{E}\ A$ is defined to be that F, otherwise it is $\lambda\_.\,\bot$. Formally, $\texttt{E}$ is defined by case analysis on $\texttt{excluded-middle}\ (\texttt{Image}\ \texttt{D}\ A)$, where $\texttt{Image}\ \texttt{D}$ is a datatype with a single constructor $\texttt{image} : (F : \texttt{Set} \to \texttt{Set}) \to \texttt{Image}\ (\texttt{D}\ F)$.

We have

$$\texttt{E}\ (\texttt{D}\ F) \equiv_{\texttt{Set}\to\texttt{Set}} F \tag{1.31}$$

for any F: because $\texttt{D}\ F$ is certainly in the image of $\texttt{D}$, we know that $\texttt{E}\ (\texttt{D}\ F)$ must be equal to G for *some* G with $\texttt{D}\ G \equiv_{\texttt{Set}} \texttt{D}\ F$, but then this G must be equal to F by injectivity of $\texttt{D}$.

Now we construct by diagonalization a `C : Set → Set` that is not in the image of `E`, thus leading to a contradiction. `C` A is defined by case analysis on `excluded-middle` (`E` A $A \equiv_{\texttt{Set}} \bot$): if `E` A A is equal to $\bot$, then `C` $A = \top$, otherwise `C` $A = \bot$.

To come to the contradiction, consider the term `B` $=$ `E (D C) (D C)`. By 1.31, we have `B` $\equiv_{\texttt{Set}}$ `C (D C)`. Is `B` equal to $\bot$ or not? By the excluded middle, there are two cases:

`B` $\equiv_{\texttt{Set}} \bot$**:** then we have `B` $\equiv_{\texttt{Set}}$ `C (D C)` $\equiv_{\texttt{Set}} \top$ by definition of `C`, but this is a contradiction with `B` $\equiv_{\texttt{Set}} \bot$.

(`B` $\equiv_{\texttt{Set}} \bot$) $\to \bot$**:** then we have `B` $\equiv_{\texttt{Set}}$ `C (D C)` $\equiv_{\texttt{Set}} \bot$ again by definition of `C`, but this is a contradiction with (`B` $\equiv_{\texttt{Set}} \bot$) $\to \bot$.

We have constructed an element of $\bot$ in the empty context, so we conclude that MLTT extended with the excluded middle and injective type constructors is inconsistent. □

In Chapter 3, we show how to postpone and reorder equations during unification without relying on the injectivity of type constructors (see in particular Lemma 3.18, Lemma 3.19, and Lemma 3.20).

1.5.3 On the importance of computation

One of the main advantages of type theory over other foundational systems such as set theory is the fact that it is a *computational* theory. This means that each term (and hence also each type) has associated computation rules. These give us a built-in way to automatically evaluate a term to its simplest form. Evaluating a term preserves definitional equality: if a (well-typed) term u evaluates to u', then $u = u'$. So to decide whether two terms are definitionally equal, we can first evaluate the two terms to normal form and then compare the normal forms.

In practice, the fact that terms can be evaluated is essential because the typechecker can do it for us. This implies we don't have to do any work to prove a definitional equality; we can just write `refl`. We already saw this when proving that `not (not b)` $\equiv_{\texttt{Bool}}$ b (1.6). After a case analysis on b, the two goal types are `not (not true)` $\equiv_{\texttt{Bool}}$ `true` and `not (not false)` $\equiv_{\texttt{Bool}}$ `false`. Since `not (not true)` evaluates to `true` and `not (not false)` evaluates to `false`, we can prove the goal with `refl` in both cases. The fact that definitional equality

can be checked *automatically* is one of the reasons why a proof assistant based on type theory is often easier to work with than one based on set theory.

Since computation is a core aspect of a proof assistant, it is important to pay special attention at the computation rules of functions defined by pattern matching. For example, we can define the boolean conjunction $\_ \mathbin{\&\&} \_ : \mathtt{Bool} \to \mathtt{Bool} \to \mathtt{Bool}$ by spelling out its four cases:

$$\begin{aligned} &\_ \mathbin{\&\&} \_ : \mathtt{Bool} \to \mathtt{Bool} \to \mathtt{Bool} \\ &\mathtt{true} \mathbin{\&\&} \mathtt{true} &&= \mathtt{true} \\ &\mathtt{true} \mathbin{\&\&} \mathtt{false} &&= \mathtt{false} \\ &\mathtt{false} \mathbin{\&\&} \mathtt{true} &&= \mathtt{false} \\ &\mathtt{false} \mathbin{\&\&} \mathtt{false} &&= \mathtt{false} \end{aligned} \tag{1.32}$$

Alternatively, we can define it more compactly with two clauses:

$$\begin{aligned} &\_ \mathbin{\&\&} \_ : \mathtt{Bool} \to \mathtt{Bool} \to \mathtt{Bool} \\ &\mathtt{true} \mathbin{\&\&} b &&= b \\ &\mathtt{false} \mathbin{\&\&} b &&= \mathtt{false} \end{aligned} \tag{1.33}$$

Not only is this definition shorter, it also gives rise to additional definitional equalities, such as $\mathtt{true} \mathbin{\&\&} b = b$. For the first definition, this equality would only be propositional.

Writing definitions by pattern matching makes it easy to see the computation rules of a function at a glance: each clause is a computation rule. In contrast, it would be less easy to see whether $\mathtt{true} \mathbin{\&\&} b = b$ had we defined $\&\&$ by eliminators as $\mathtt{elim}_{\mathtt{Bool}}\ (\mathtt{Bool} \to \mathtt{Bool})\ (\lambda b.\ b)\ (\lambda b.\ \mathtt{false})$. Because definitions by pattern matching follow the principle of 'what you see is what you get', the user can be confident which equalities hold by definition and which ones he has to prove himself.

When translating definitions by pattern matching to eliminators, we have to make sure that the translated definition still satisfies the same computation rules. This is not at all straightforward: at each step of the translation (Section 4.3) there is a risk that we translate one construct to a different one that has the same type but not the same computation rules. In particular, the most general general unifier computed by the unification algorithm in Chapter 3 needs to have the right computational behaviour. If this is the case, we call it a *strong unifier* (Definition 3.45). For each of the unification rules except the deletion rule, we prove that they are strong (Lemma 3.43, Lemma 3.44, and Lemma 3.61).

1.6 Overview and contributions

In the previous sections, we touched on many of the subjects in this thesis concerning dependent pattern matching and proof-relevant unification. Here follows an overview of the contents of the remaining chapters.

Chapter 2 presents Agda Lite, a minimal dependently typed language with indexed datatypes and dependent pattern matching. It describes how to check definitions by dependent pattern matching by translating them to a structurally recursive *case tree*. In particular, it presents a simple first-order unification algorithm to check case splits on elements of indexed datatypes. The main contribution of this chapter consists of three new criteria on definitions by dependent pattern matching that, when satisfied, ensure that dependent pattern matching is conservative over standard type theory. The untyped unification algorithm in this chapter with these three restrictions is a conservative approximation of the proof-relevant unification algorithm in Chapter 3, but it has the advantage that it is easier to understand and simpler to add to an existing implementation of dependent pattern matching without replacing the whole unification algorithm.

Chapter 3 presents a new *typed* and *proof-relevant* unification algorithm that can be used in place of the untyped algorithm in the previous chapter for checking definitions by dependent pattern matching. Our main innovation is to represent a most general unifier as an *equivalence* between two unification problems, serving as evidence of its correctness. These unification problems are represented as telescopic equalities where each equation can depend on the previous ones, which has the following three advantages:

- We avoid the use of heterogeneous equality by McBride (2000), which requires the K axiom to use.
- We can give more general unification rules for indexed datatypes that solve multiple equations at once.
- Because we keep track of dependencies between equations, we can safely postpone and reorder them without running into paradoxes.

To guarantee good computational properties of the unifiers produced by our unification algorithm, we introduce the notion of a *strong unification rule*. We show that all the unification rules used by our algorithm are strong ones, except for the deletion rule.

We give two extensions to the unification algorithm: a simple one to deal with η-equality of record types, and a more involved one to deal with injectivity of

constructors of indexed datatypes in general, which we call *higher-dimensional unification*. We also describe our implementation of this unification algorithm that replaces the old algorithm used by Agda for checking definitions by dependent pattern matching, eliminating the previous ad-hoc restrictions and fixing a number of bugs in the process.

Chapter 4 makes good on the promises in the previous two chapters by showing how everything in them can be implemented in terms of standard datatype eliminators. In particular, we present the five basic constructions of McBride, Goguen, and McKinna (2006): case analysis, structural recursion, injectivity, disjointness, and acyclicity; using homogeneous telescopic equalities rather than heterogeneous ones. We also present two techniques that are useful when applying these constructions: *basic analysis* and *specialization by unification*. In the proof of our main theorem (Theorem 4.27) we apply these techniques to translate an arbitrary valid definition by dependent pattern matching to eliminators. Moreover, if all the rules used by the unification algorithm are strong, we can guarantee that all clauses are preserved as definitional equalities during this equation (Theorem 4.29).

Finally, **Chapter 5** wraps up the thesis and looks forward to the future of dependent pattern matching with a number of possible directions for further research.

In more detail, I make the following technical contributions:

- I present a set of three restrictions to dependent pattern matching as described by Coquand (1992) that make it safe to use in any (consistent) extension of MLTT. These restrictions are strictly more general than previous attempts. In contrast to previous attempts, it is not based on a syntactic check, but on a restriction to the unification algorithm used for pattern matching (Section 2.3).
- I implement these three restrictions as a patch to Agda. I compare it to the old criterion offered by Agda for pattern matching without K on the grounds of adequacy, soundness, and generality. As of Agda version 2.4.0, my implementation replaces the old version of `--without-K` (Section 2.3.3).
- I give a new representation of unification rules and most general unifiers in a dependently typed setting as *equivalences* between solution spaces represented by telescopic systems of equations. This gives a general way to characterize soundness of unification rules internally to the underlying type theory (Section 3.1).

- I phrase the unification rules of Goguen et al. (2006) as equivalences and show how to implement them in terms of eliminators (Section 3.2).
- I present new unification rules for indexed families of data types that work on heterogeneous equations and can solve multiple equations at once, making them more general than the ones in our previous work (Cockx, Devriese, and Piessens, 2014b) (Section 3.2.2).
- I describe new unification rules that deal with eta-equality of values of record type and show how they fit in the framework of unifiers as equivalences (Section 3.2.3).
- I define the notion of a *strong unification rule*, a unification rule which has certain good computational properties as a term in type theory. I also prove that all of the unification rules in this thesis are in fact strong, except for the deletion rule (Section 3.3).
- I show how to make the injectivity rule for indexed datatypes more general by generalizing over the indices in the type of the equation, generating new equations between equality proofs (called *higher-dimensional equations*) in the process (Section 3.4.2).
- I show how to apply regular unification rules to higher-dimensional equations by *lifting* them to a higher dimension (Section 3.4.3).
- I show how higher-dimensional unification formalizes the concept of forced constructor arguments, a heuristic that allows unification to skip certain constructor arguments if they are determined by the type of the constructor (Corollary 3.56).
- I reimplement the unification algorithm used by Agda for pattern matching on indexed families of data types based on my framework for proof-relevant unification, fixing a number of bugs in the process and making it more amenable to future extensions. This new unification algorithm has been released as part of Agda version 2.5.1[3] (Section 3.5).
- I give a formal proof that definitions by pattern matching that have been checked using our unification algorithm are conservative over MLTT by translating them to eliminators in the style of Goguen et al. (2006), but *without* relying on K (Section 4.3).
- I also prove that in case the unification algorithm produces strong unifiers, then the translated function still satisfies the same computation rules as the original (Section 4.3.1).

[3] Available from `http://wiki.portal.chalmers.se/agda/`.

Most of these results were already presented in our earlier work (Cockx, Devriese, and Piessens, 2014b, 2016a,b; Cockx and Devriese, 2017), of which I was the main author and contributor. My contributions over the previously published versions of this work are the following:

- I give a more general restriction to the injectivity rule in the criterion for pattern matching without K that allows skipping the unification of forced constructor arguments (Definition 2.29). In particular, this allows Example 2.31 to be accepted, which was not possible with the previous version of the restriction (Section 2.3.1).
- I give a formal proof that (a suitably internalized notion of) most general unifiers is really equivalent to equivalences (Lemma 3.11 and Lemma 3.12 in Section 3.1.2).
- I give a new definition of a strong unifier (Definition 3.45). Compared the previous definition (Cockx et al., 2016a), this definition is more natural to work with and allows me to prove Theorem 4.29, while it is still satisfied by all the unification rules (Section 3.3).
- I prove that lifting a strong unifier results again in a strong unifier (Lemma 3.61 in Section 3.3).
- I give a more detailed proof of Theorem 4.29, in particular I prove Lemma 4.13 about the computational behaviour of the function $\texttt{below}_{\texttt{D}}$ used in the translation (Section 4.1.2).
- The presentation of many examples, definitions, lemmas and theorems was improved compared to the paper versions.

Our work has already made a significant impact on other people working with dependent types:

- Our work is used on a daily basis by all users of Agda, one of the most popular dependently typed programming languages currently in existence.
- Our work has heavily influenced the development of the new version of Equations, a plugin for using dependent pattern matching in Coq (Mangin and Sozeau, 2017).
- The authors of the Lean theorem prover developed at Microsoft Research cite our work as an important source of inspiration for the implementation of the Lean system (de Moura, Kong, Avigad, van Doorn, and von Raumer, 2015).

To our knowledge, the latter two systems are based on the older version of our work (Cockx et al., 2014b) and do not yet include newer improvements such as higher-dimensional unification (Section 3.4).

Chapter 2

Dependent pattern matching

> *The universe has a pattern. The further we push ourselves to transcend our limitations, the greater we perceive that pattern to be. It's as though it was all a great game, but you only discover the rules by playing...*
>
> — Sarah Newton (2016)

Dependent pattern matching is an intuitive way to write programs and proofs in dependently typed languages. It is reminiscent of both pattern matching in functional languages and case analysis in on-paper mathematics. However, the original version of dependent pattern matching by Coquand (1992) is incompatible with new type theories such as homotopy type theory. As a consequence, it cannot be used in those theories so their proofs are typically harder to write and to understand. The source of this incompatibility is the reliance of dependent pattern matching on the K rule – also known as the uniqueness of identity proofs – which is inadmissible in HoTT.

In this chapter, we propose a criterion for checking whether a given definition by pattern matching is safe to use in a theory without K, thus bringing the benefits of dependent pattern matching to languages such as HoTT. To study dependent pattern matching we introduce Agda Lite, a minimal dependently typed language with support for dependent pattern matching (Section 2.1). We also show how definitions by dependent pattern matching in Agda Lite can be checked by translating them to a case tree (Section 2.2). The main contribution of this chapter is a general criterion for dependent pattern matching without K (Section 2.3). Finally, we discuss related work (Section 2.4).

2.1 Agda Lite: a minimal language with dependent pattern matching

In this section, we introduce Agda Lite, a minimal version of Martin-Löf type theory extended with inductive families of datatypes and dependent pattern matching. The purpose of this language is to be as simple as possible for the study of dependent pattern matching, while still being a valid subset of the full Agda language. The goal is *not* to give a full description of a possible core language for Agda, though such as effort would be worthwhile on its own.

2.1.1 Basic syntax, typing, and evaluation rules

The basis of Agda Lite is Martin-Löf's Intuitionistic Theory of Types with dependent function types, inductive families, and universes (Martin-Löf, 1972; Martin-Löf, 1984). However, the results in this thesis should be equally applicable in other type theories with inductive families such as the Unified Theory of Dependent Types (UTT) by Luo (1994) or the Calculus of Inductive Constructions (CIC) used by Coq. The main reason we don't use these more expressive calculi is because we don't need their additional features, and not using them means our results are applicable to any type theory that includes at least the typing rules of Martin-Löf type theory.

Syntax of Agda Lite

Types and terms share the same syntactic class. As a convention, types are indicated by capital letters A, B, ... and other terms by small letters u, v,... Aside from the standard type-theoretic constructs, the syntax includes data types `D`, constructors `c`, and defined functions `f`.

Definition 2.1 (Types and terms)**.**

$$
\begin{array}{rcll}
A, B, u, v & ::= & x & \text{(variables)} \\
& | & u\ v & \text{(application)} \\
& | & \lambda x.\, u & \text{(lambda abstraction)} \\
& | & (x : A) \to B & \text{(dependent function type)} \\
& | & \texttt{Set}_\ell & \text{(universe } \ell\text{)} \\
& | & \texttt{D} & \text{(datatype)} \\
& | & \texttt{c} & \text{(data constructor)} \\
& | & \texttt{f} & \text{(defined function)}
\end{array}
\tag{2.1}
$$

Definition 2.2 (Free and bound variables)**.** An occurrence of a variable x is *bound* if it occurs in u in an expression of the form $\lambda x.\, u$ or in B in an expression of the form $(x : A) \to B$, otherwise it is *free*. A variable is bound (respectively free) in an expression if at least one of its occurrences is bound (respectively free). The set of variables that occur freely in u is indicated by $FV(u)$.

As a convention, we never distinguish terms if they are equal up to α-renaming of bound variables.

Definition 2.3 (Simultaneous substitution)**.** Substitutions σ, τ, ... are of the form $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$. The empty substitution is written as $[]$ and the composition of two substitutions is written as $\sigma;\tau$ (first apply σ, then τ). Simultaneous substitution $u[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ is defined by simultaneously replacing all free occurrences of $x_1, \ldots, x_n$ in u by $v_1, \ldots, v_n$, avoiding variable capture by renaming bound variables when necessary.

A *context* contains the names and types of the free variables in an expression. Contexts are indicated by Greek capitals $\Gamma, \Delta, \ldots$

Definition 2.4 (Context)**.**

$$\begin{array}{rcll} \Gamma & ::= & () & \text{(empty context)} \\ & | & \Gamma(x : A) & \text{(context extension)} \end{array} \tag{2.2}$$

The syntax $()$ for an empty context is usually omitted unless it occurs on its own. If $(x : A)$ occurs somewhere in Γ, then we write $(x : A) \in \Gamma$.

Basic typing rules

As standard in type theory, we present the type system of Agda Lite in the style of natural deduction, i.e. by giving typing rules of the form

$$\frac{J_1 \qquad \ldots \qquad J_n}{J}$$

where $J_1, \ldots, J_n$ and J are *judgements* representing the hypotheses and the conclusion of the rule. A judgement is a meta-level statement about the language under study, in this case Agda Lite. The primary forms of judgement of Agda Lite are the following:

Γ **context** , meaning Γ is a valid context.

$$\frac{}{() \ \textbf{context}}$$

$$\frac{\Gamma \vdash A : \texttt{Set}_\ell \qquad x \notin FV(\Gamma)}{\Gamma(x : A) \ \textbf{context}}$$

Figure 2.1: The rules for valid contexts in Agda Lite.

$\Gamma \vdash u : A$, meaning that the term u has type A in context Γ.

$\Gamma \vdash u_1 = u_2 : A$, meaning that u_1 is definitionally equal to u_2 of type A in context Γ.

Some rules also contain side conditions next to their hypotheses, such as $x \notin FV(\Gamma)$ in the rule for context extension. We write $u : A$ for $\Gamma \vdash u : A$ if Γ is clear from the context and $u_1 = u_2$ for $\Gamma \vdash u_1 = u_2 : A$ if both Γ and A are clear from the context.

The basic rules for context validity, typing, and definitional equality are given in Figure 2.1, Figure 2.2, and Figure 2.3 respectively. These rules contain metavariables Γ for contexts and $A, B, u, v, \ldots$ for types and terms. Replacing these metavariables by actual contexts and terms respectively results in an instance of the rule. A judgement is *derivable* if it occurs as the conclusion of such an rule instance, and all of the hypotheses of the rule instance are derivable as well.

Telescopes

A telescope is a list of typed variable bindings where each type can depend on the previous variables.

Definition 2.5 (Telescope)**.** Telescopes are defined by the following grammar:

$$\begin{array}{rcll} \Delta & ::= & () & \text{(empty telescope)} \\ & | & (x : A)\Delta & \text{(telescope extension)} \end{array} \tag{2.3}$$

Like for contexts, we usually skip the syntax () for an empty telescope unless it occurs on its own. For example, $(m : \mathbb{N})(p : m \equiv_{\mathbb{N}} \texttt{zero})$ is a telescope of length 2.

Telescopes are much like contexts in the sense that they consist of a sequence of variable typings of the form $(x : A)$. However, they are used for different

$$\frac{\Gamma\ \textbf{context} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash u : A_1 \qquad \Gamma \vdash A_1 = A_2 : \mathtt{Set}_\ell}{\Gamma \vdash u : A_2}$$

$$\frac{\Gamma\ \textbf{context}}{\Gamma \vdash \mathtt{Set}_\ell : \mathtt{Set}_{\ell+1}}$$

$$\frac{\Gamma \vdash A : \mathtt{Set}_\ell \qquad \Gamma(x : A) \vdash B : \mathtt{Set}_{\ell'}}{\Gamma \vdash (x : A) \to B : \mathtt{Set}_{\max(\ell,\ell')}}$$

$$\frac{\Gamma(x : A) \vdash u : B}{\Gamma \vdash \lambda x.\ u : (x : A) \to B}$$

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash f\ u : B[x \mapsto u]}$$

Figure 2.2: The core typing rules of Agda Lite, including dependent function types $(x : A) \to B$ and an infinite hierarchy of universes $\mathtt{Set}_0, \mathtt{Set}_1, \mathtt{Set}_2, \ldots$

purposes so it is best to keep the two concepts separate. While contexts grow to the right, telescopes grow to the left. One way to think about a telescope is as the *tail* of a context: while a context must always be closed, a telescope can contain free variables from an ambient context, and the telescope can be added to that context to produce a new, extended context.

If there are multiple variables of the same type after each other, then we usually only write the type once. For example, $(x\ y\ z : A)$ stands for the telescope $(x : A)(y : A)(z : A)$.

Telescopes are used as the type of a list of terms. A list of terms is indicated by a bar above the letter, for example $\bar{t} = (\mathtt{zero}; \mathtt{refl}) : (m : \mathbb{N})(p : m \equiv_{\mathbb{N}} \mathtt{zero})$. We also write $()$ for the empty list of terms. The typing rules for telescopes and lists of terms are given in Figure 2.4.

Telescopes are useful for various other purposes: a telescope can be used

- ... as an extension to the context: $\Gamma\Delta$ is defined by $\Gamma() := \Gamma$ and $\Gamma((x : A)\Delta) := (\Gamma(x : A))\Delta$. In particular, if Δ is a valid telescope in the empty context then Δ can be used as the context $()\Delta$.

$$\frac{\Gamma(x : A) \vdash u : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda x.\ u)\ v = u[x \mapsto v] : B[x \mapsto v]}$$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash u = u : A}$$

$$\frac{\Gamma \vdash u_1 = u_2 : A}{\Gamma \vdash u_2 = u_1 : A}$$

$$\frac{\Gamma \vdash u_1 = u_2 : A \qquad \Gamma \vdash u_2 = u_3 : A}{\Gamma \vdash u_1 = u_3 : A}$$

$$\frac{\Gamma \vdash u_1 = u_2 : A_1 \qquad \Gamma \vdash A_1 = A_2 : \mathtt{Set}_\ell}{\Gamma \vdash u_1 = u_2 : A_2}$$

$$\frac{\Gamma \vdash A_1 = A_2 : \mathtt{Set}_\ell \qquad \Gamma(x : A_1) \vdash B_1 = B_2 : \mathtt{Set}_{\ell'}}{\Gamma \vdash (x : A_1) \to B_1 = (x : A_2) \to B_2 : \mathtt{Set}_{\max(\ell,\ell')}}$$

$$\frac{\Gamma(x : A) \vdash u_1 = u_2 : B}{\Gamma \vdash \lambda x.\ u_1 = \lambda x.\ u_2 : (x : A) \to B}$$

$$\frac{\Gamma \vdash f_1 = f_2 : (x : A) \to B \qquad \Gamma \vdash u_1 = u_2 : A}{\Gamma \vdash f_1\ u_1 = f_2\ u_2 : B[x \mapsto u_1]}$$

Figure 2.3: The rules for definitional equality in Agda Lite, including rules for β-equality, reflexivity, symmetry, transitivity, and congruence.

- ... as the names of the variables of a parallel substitution: $u[\Delta \mapsto \bar{v}]$ is defined by substituting the values $\bar{v}$ for the variables of Δ in u.
- ... as the argument types of an iterated function type: $\Delta \to B$ is defined by $() \to B := B$ and $(x : A)\Delta \to B := (x : A) \to (\Delta \to B)$.
- ... in the definition of an iterated lambda abstraction: $\lambda\Delta.\ u$ is defined by $\lambda().\ u := u$ and $\lambda((x : A)\Delta).u := \lambda x.\ (\lambda\Delta.\ u)$.
- ... as a list of the variables in the telescope: $f\ \Delta$ is defined by $f\ () := f$ and $f\ ((x : A)\Delta) := (f\ x)\ \Delta$.

These various interpretations of telescopes can be used together. For example, if $\Gamma \vdash f : \Delta \to B$ then we have $\Gamma\Delta \vdash f\ \Delta : B$.

$$\frac{\Gamma \textbf{ context}}{\Gamma \vdash () \textbf{ telescope}}$$

$$\frac{\Gamma \vdash A : \texttt{Set}_\ell \qquad \Gamma(x : A) \vdash \Delta \textbf{ telescope}}{\Gamma \vdash (x : A)\Delta \textbf{ telescope}}$$

$$\frac{\Gamma \textbf{ context}}{\Gamma \vdash () : ()}$$

$$\frac{\Gamma \vdash u : A \qquad \Gamma \vdash \bar{u} : \Delta[x \mapsto u]}{\Gamma \vdash (u; \bar{u}) : (x : A)\Delta}$$

Figure 2.4: The typing rules for telescopes. The substitution $\Delta[x \mapsto u]$ substitutes u for x in the types of Δ, not the variables.

If $\bar{t} : \Delta_1\Delta_2\Delta_3$ then $\bar{t}|_{\Delta_2}$ stands for the sublist of $\bar{t}$ corresponding to the telescope Δ_2. If $\Gamma \vdash \bar{t} : \Delta_1\Delta_2\Delta_3$, then we have $\Gamma \vdash \bar{t}|_{\Delta_2} : \Delta_2[\Delta_1 \mapsto \bar{t}|_{\Delta_1}]$.

A function between telescopes is called a *telescope mapping*. A telescope mapping $f : \Delta \to \Delta'$ maps variables of type Δ to values of type Δ'. Telescope mappings generalize the concept of a (non-dependent) function to multiple inputs and multiple outputs. They could be encoded as normal functions by representing a telescope by an iterated Σ-type, but we find it useful to define them as a first-class concept.

Another way to view a telescope mapping $f : \Delta \to \Delta'$ is as a *typed* variant of a substitution. In particular, if we have a term $u : A$ with free variables coming from Δ', then we can apply the substitution $[\Delta' \mapsto f\ \Delta]$ to it, replacing the variables from Δ' by the values given by $f\ \Delta$, to get a term u' with free variables coming from Δ.

Example 2.6. Suppose $\Delta = (k : \mathbb{N})$ and $\Delta' = (m\ n : \mathbb{N})$ and let $f\ k = (\texttt{zero}; \texttt{suc}\ k)$. We have $\Delta' \vdash m + n : \mathbb{N}$, so applying the substitution $[\Delta' \mapsto f\ \Delta] = [m \mapsto \texttt{zero}; n \mapsto \texttt{suc}\ k]$ gives us $\Delta \vdash \texttt{zero} + \texttt{suc}\ k : \mathbb{N}$.

When we use a telescope mapping $f : \Delta \to \Delta'$ as a substitution, the substitution goes in the 'opposite' direction: it takes terms with free variables Δ' to terms with free variables Δ. In this case, the type of f is often written as $\Delta \vdash f : \Delta'$. However, since in this thesis we use telescope mappings mainly as functions rather than as substitutions, we stick to the notation $f : \Delta \to \Delta'$.

Declarations

An Agda Lite program consists of a sequence of *declarations*. A declaration is either a datatype declaration with zero or more constructor declarations (Section 2.1.2) or a function declaration with one or more clauses (Section 2.1.3).

Definition 2.7 (Declaration)**.**

$$
\begin{array}{rcll}
\mathit{decl} & ::= & \textbf{data}\ \mathtt{D}\ \Gamma : \Xi \to \mathtt{Set}_\ell\ \textbf{where}\ \mathit{con}^* & \text{(datatype declaration)} \\
& \mid & \mathtt{f} : A\ \mathit{cls}^+ & \text{(function declaration)} \\
\\
\mathit{con} & ::= & \mathtt{c} : \Delta \to \mathtt{D}\ \bar{u} & \text{(constructor declaration)} \\
\\
\mathit{cls} & ::= & \mathtt{f}\ \mathit{pat}^* = u & \text{(clause)} \\
& \mid & \mathtt{f}\ \mathit{pat}^* \pitchfork x & \text{(absurd clause)}
\end{array}
\tag{2.4}
$$

An absurd clause $\mathtt{f}\ \mathit{pat}^* \pitchfork x$ allows the right-hand side to be skipped if the type of the variable x is empty. In the full Agda language, the concrete syntax for an absurd clause is $\mathtt{f}\ \bar{p}$ where the pattern variable x in $\bar{p}$ has been replaced by an *absurd pattern* ().

Definition 2.8 (Pattern, underlying term)**.**

$$
\begin{array}{rcll}
\mathit{pat} & ::= & x & \text{(pattern variable)} \\
& \mid & \mathtt{c}\ \mathit{pat}^* & \text{(constructor pattern)} \\
& \mid & .u & \text{(inaccessible pattern)}
\end{array}
\tag{2.5}
$$

For each pattern p, we also define the *underlying term* $\lceil p \rceil$ as follows:

$$
\begin{array}{lcl}
\lceil x \rceil & = & x \\
\lceil \mathtt{c}\ p_1\ \ldots\ p_n \rceil & = & \mathtt{c}\ \lceil p_1 \rceil\ \ldots \lceil p_n \rceil \\
\lceil .u \rceil & = & u
\end{array}
\tag{2.6}
$$

2.1.2 Inductive families of datatypes

Inductive families of datatypes are (dependent) types inductively defined by a number of constructors (Dybjer, 1991). Inductive families can also have *parameters* and *indices*.

Example 2.9. $\mathbb{N}$ is defined as an inductive datatype with the constructors `zero` and `suc`:

$$
\begin{array}{l}
\textbf{data}\ \mathbb{N} : \mathtt{Set}\ \textbf{where} \\
\quad \mathtt{zero} : \mathbb{N} \\
\quad \mathtt{suc}\ \ : \mathbb{N} \to \mathbb{N}
\end{array}
\tag{2.7}
$$

Example 2.10. $A \uplus B$ is an inductive datatype with two parameters A and B and two constructors `left` and `right`:

$$\begin{aligned}&\textbf{data}\ A \uplus B : \mathtt{Set}\ \textbf{where}\\ &\quad \mathtt{left}\ \ : A \to A \uplus B\\ &\quad \mathtt{right} : B \to A \uplus B\end{aligned} \tag{2.8}$$

Example 2.11. `Vec` $A\ n$ is an inductive family with one parameter $A : \mathtt{Set}$, one index $n : \mathbb{N}$, and two constructors `nil` and `cons`:

$$\begin{aligned}&\textbf{data}\ \mathtt{Vec}\ A : \mathbb{N} \to \mathtt{Set}\ \textbf{where}\\ &\quad \mathtt{nil}\ \ : \mathtt{Vec}\ A\ \mathtt{zero}\\ &\quad \mathtt{cons} : (n : \mathbb{N}) \to A \to \mathtt{Vec}\ A\ n \to \mathtt{Vec}\ A\ (\mathtt{suc}\ n)\end{aligned} \tag{2.9}$$

Example 2.12. The type $m \leq n$ is an inductive family with two indices and two constructors `lz` and `ls`:

$$\begin{aligned}&\textbf{data}\ \_ \leq \_ : \mathbb{N} \to \mathbb{N} \to \mathtt{Set}\ \textbf{where}\\ &\quad \mathtt{lz} : (n : \mathbb{N}) \to \mathtt{zero} \leq n\\ &\quad \mathtt{ls} : (m\ n : \mathbb{N}) \to m \leq n \to \mathtt{suc}\ m \leq \mathtt{suc}\ n\end{aligned} \tag{2.10}$$

Example 2.13. The identity type $x \equiv_A y$ can also be defined as an inductive family. We follow the definition of the identity type by Paulin-Mohring (1993), as an inductive family with two parameters $A : \mathtt{Set}_\ell$ and $x : A$, one index $y : A$, and one constructor `refl`:

$$\begin{aligned}&\textbf{data}\ x \equiv_A \_ : A \to \mathtt{Set}_\ell\ \textbf{where}\\ &\quad \mathtt{refl} : x \equiv_A x\end{aligned} \tag{2.11}$$

In the original definition of indexed families by Dybjer (1991), parameters are required to occur uniformly everywhere in the definition of the datatype, while indices can vary from constructor to constructor. Agda is less restrictive and also allows parameters to occur non-uniformly in the types of recursive constructor arguments, but not in their return types. The work in this thesis is valid for the more liberal definition used by Agda, and hence also for the original definition.

Definition 2.14 (Datatype validity)**.** Consider a (well-scoped) declaration with parameters Δ and indices Ξ of the form

$$\begin{aligned}&\textbf{data}\ \mathtt{D}\ \Delta : \Xi \to \mathtt{Set}_\ell\ \textbf{where}\\ &\quad \mathtt{c}_1\ \ : \Delta_1 \to \mathtt{D}\ \bar{u}_1\\ &\qquad \vdots\\ &\quad \mathtt{c}_n\ \ : \Delta_n \to \mathtt{D}\ \bar{u}_n\end{aligned} \tag{2.12}$$

This datatype declaration is valid on the following conditions:

- Δ is a valid context.
- Ξ is a valid telescope in context Δ.
- For each $i = 1, \ldots, n$, Δ_i is a valid telescope in context Δ, assuming $\mathtt{D} : \Delta \to \Xi \to \mathtt{Set}_\ell$.[1]
- All types in Δ_i are of level ℓ or lower.
- For each $i = 1, \ldots, n$, $\bar{u}_i$ is of the form $\bar{x}; \bar{u}'_i$ where $\bar{x}$ are the variables bound in Δ and $\bar{u}'_i$ is of type Ξ in context $\Delta\Delta_i$.
- $\mathtt{D}$ occurs *strictly positively* in the type of the arguments of each constructor, i.e. for each i and each $(x_{ij} : B_{ij}) \in \Delta_i$, B_{ij} either doesn't contain $\mathtt{D}$ or is of the form $\Phi \to \mathtt{D}\ \bar{v}_{ij}$ where Φ doesn't contain $\mathtt{D}$.
- None of the types in Δ_i are dependent on the recursive variables in Δ_i, i.e. those variables with a type of the form $\Phi \to \mathtt{D}\ \bar{v}_{ij}$.

For a valid data declaration of the form above, we add the following typing rules:

$$\frac{\Gamma\ \textbf{context}}{\Gamma \vdash \mathtt{D} : \Delta \to \Xi \to \mathtt{Set}_\ell}$$

$$\frac{\Gamma \vdash \bar{r} : \Delta}{\Gamma \vdash \mathtt{c_i} : \Delta_i[\Delta \mapsto \bar{r}] \to \mathtt{D}\ \bar{r}\ \bar{u}_i[\Delta \mapsto \bar{r}]}$$

These typing rules can be used to type the subsequent declarations of the Agda Lite program.

The values of the parameters $\bar{r}$ are not arguments to the constructor $\mathtt{c}$, not even implicitly. This is intentional: requiring constructors to remember their parameters is impractical from an implementation perspective, so we make sure they are never needed for the algorithms described in this thesis.

When we say something about a generic datatype $\mathtt{D}$ in this thesis, we will consider $\mathtt{D}$ to be already applied to its parameters Δ, so we have $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$. For example, we consider $\mathtt{D}$ to stand for $\mathtt{Vec}\ A$ rather than just $\mathtt{Vec}$.

We write $\overline{\mathtt{D}}$ for the telescope $(\bar{u} : \Xi)(x : \mathtt{D}\ \bar{u})$, for example $\overline{\mathtt{Vec}} = (n : \mathbb{N})(x : \mathtt{Vec}\ A\ n)$.

Mutually defined datatypes are not supported directly by Agda Lite but can be encoded as a single datatype with an additional index.

[1] If we only assume $\mathtt{D}\ \Delta : \Xi \to \mathtt{Set}_\ell$ here, we get the more restrictive version of datatypes where parameters must occur uniformly everywhere.

Example 2.15. Consider two mutually defined indexed datatypes `Even` and `Odd` defined as follows:

```
data Even : ℕ → Set where
  even-zero : Even zero
  even-suc  : (n : ℕ) → Odd n → Even (suc n)

data Odd : ℕ → Set where
  odd-suc : (n : ℕ) → Even n → Odd (suc n)
```
(2.13)

In Agda Lite, these can be encoded as a single datatype as follows:

```
data EvenOdd : Bool → ℕ → Set where
  even-zero : EvenOdd true zero
  even-suc  : (n : ℕ) → EvenOdd false n → EvenOdd true (suc n)
  odd-suc   : (n : ℕ) → EvenOdd true n → EvenOdd false (suc n)
```
(2.14)

where `Even` = `EvenOdd true` and `Odd` = `EvenOdd false`.

On the other hand, inductive-recursive datatypes (Dybjer, 2000) and inductive-inductive datatypes (Forsberg and Setzer, 2010) go beyond the scope of this thesis.

In most presentations of inductive datatypes in type theory, each datatype is equipped with a *datatype eliminator*. This eliminator allows us to define functions by case analysis and induction on elements of the datatype. Agda Lite instead allows the user to define functions by dependent pattern matching. This allows the user to define their own induction principles. In particular, the standard datatype eliminator can be defined by pattern matching.

2.1.3 Definitions by dependent pattern matching

Dependent pattern matching is a way to define new functions by specifying a number of clauses the function has to satisfy. The behaviour of a function by pattern matching when applied to some term u is determined by *matching* u against the patterns of each of the clauses. The definitions in this section are based on the ones given by Goguen et al. (2006).

Definition 2.16 (Pattern matching)**.** Matching a term u against a pattern p can either produce a substitution σ for the pattern variables of p ($\textsc{match}(p, u) \Rightarrow \sigma$), produce a conflict ($\textsc{match}(p, u) \Rightarrow \bot$), or it can get stuck. See Figure 2.5 for the rules involving matching.

$$\frac{}{\textsc{match}(x, t) \Rightarrow [x \mapsto t]}$$

$$\frac{\textsc{match}(\bar{p}, \bar{t}) \Rightarrow \sigma}{\textsc{match}(\mathtt{c}\ \bar{p}, \mathtt{c}\ \bar{t}) \Rightarrow \sigma} \qquad \frac{\textsc{match}(\bar{p}, \bar{t}) \Rightarrow \bot}{\textsc{match}(\mathtt{c}\ \bar{p}, \mathtt{c}\ \bar{t}) \Rightarrow \bot}$$

$$\frac{\mathtt{c}_1 \neq \mathtt{c}_2}{\textsc{match}(\mathtt{c}_1\ \bar{p}, \mathtt{c}_2\ \bar{t}) \Rightarrow \bot}$$

$$\frac{}{\textsc{match}((), ()) \Rightarrow []}$$

$$\frac{\textsc{match}(p, t) \Rightarrow \sigma \qquad \textsc{match}(\bar{p}, \bar{t}) \Rightarrow \sigma'}{\textsc{match}((p; \bar{p}), (t; \bar{t})) \Rightarrow \sigma; \sigma'}$$

$$\frac{\textsc{match}(p, t) \Rightarrow \bot}{\textsc{match}((p; \bar{p}), (t; \bar{t})) \Rightarrow \bot} \qquad \frac{\textsc{match}(\bar{p}, \bar{t}) \Rightarrow \bot}{\textsc{match}((p; \bar{p}), (t; \bar{t})) \Rightarrow \bot}$$

Figure 2.5: Rules describing the pattern matching algorithm.

Consider a (well-scoped) function definition of the form

$$\begin{aligned} &\mathtt{f} : \Delta \to T \\ &cl_1 \\ &\vdots \\ &cl_n \end{aligned} \tag{2.15}$$

where each clause cl_i is either of the form $\mathtt{f}\ \bar{p}_i = u_i$ or $\mathtt{f}\ \bar{p}_i \Cap x$. For this definition to make sense, it needs to satisfy at least the following properties:

- $\Delta \to T$ is a valid type in the empty context.
- For each clause $\mathtt{f}\ \bar{p}_i = u_i$ or $\mathtt{f}\ \bar{p}_i \Cap x$, there exists some telescope Δ_i such that $\Delta_i \vdash \lceil \bar{p}_i \rceil : \Delta$.
- Each variable in Δ_i occurs exactly once in an accessible position in $\bar{p}_i$.
- For each regular clause $\mathtt{f}\ \bar{p}_i = u_i$, we have $\Delta_i \vdash u_i : T[\Delta \mapsto \lceil \bar{p}_i \rceil]$.
- For each absurd clause $\mathtt{f}\ \bar{p}_i \Cap x$, the type of x in Δ_i is an empty type.

Aside from these requirements, there are also a number of global requirements on the clauses for the definition of $\mathtt{f}$ to be valid:

Completeness For each list of closed terms $\bar{s} : \Delta$, there must be a pattern $\bar{p}$ such that $\bar{s}$ matches $\bar{p}$. This is required to have canonicity, i.e. that any closed normal form of an inductive family is constructor-headed.

Termination There can be no $\bar{s} : \Delta$ such that there is an infinite sequence of evaluation steps starting from $f\ \bar{s}$. This is required to have strong normalization, i.e. that any sequence of evaluation steps starting from a well-typed term eventually ends in a normal form.

Confluence If $\bar{s}$ matches more than one pattern in the definition of `f`, any choice of which clause to apply should lead to the same result. This is required to have the Church-Rosser property, i.e. all normal forms of a term are definitionally equal.

In the next two sections, these three requirements will be strengthened further. For now, suppose that the definition of `f` is valid, then we add the following typing rule:

$$\frac{\Gamma\ \textbf{valid}}{\Gamma \vdash \texttt{f} : \Delta \to T}$$

and for each non-absurd clause $\texttt{f}\ \bar{p}_i = u_i$ of `f` the rule

$$\frac{\Gamma \vdash \bar{v} : \Delta \qquad \textsc{match}(\bar{p}_i, \bar{v}) \Rightarrow \sigma}{\Gamma \vdash \texttt{f}\ \bar{v} = u_i\sigma : T[\Delta \mapsto \bar{v}]}$$

2.2 Checking definitions by dependent pattern matching

At first sight, the three properties of functions by pattern matching we required above (completeness, termination, and confluence) seem to be everything needed to guarantee the 'good' properties of type theory, in particular consistency of the theory and decidability of typechecking. However, there are at least three reasons to impose stricter requirements:

- The typechecker needs to *check* whether these properties hold of a given definition. This is not straightforward to do for their current formulation.
- When new axioms are added to the theory, these three properties are no longer enough to guarantee soundness. For example, the definition of K (1.29) is complete, terminating, and confluent, but it is unsound in combination with univalence.

- To evaluate functions by pattern matching *efficiently*, we need a fast way to determine which clause matches a particular combination of arguments. This is especially important when the number of clauses becomes large.

For the three reasons outlined above, definitions by pattern matching are usually represented as a *case tree* (Augustsson, 1985). If a function can be represented by a case tree, its clauses are automatically complete and non-overlapping, so it satisfies completeness and confluence. A case tree also allows us to evaluate the function efficiently, without looking at each clause in sequence. Finally and most importantly for the purposes of this thesis, functions representable by a case tree can be translated to datatype eliminators, thus guaranteeing their conservativity over a more basic type theory. This is the subject of Chapter 4.

In this section, we show how to check definitions by dependent pattern matching in Agda Lite. First, we show how case trees are built from successive case splits (Section 2.2.1). We also give a simple termination criterion: functions should be structurally recursive (Section 2.2.2). To deal with case splitting on indexed datatypes, we apply unification to the indices (Section 2.2.3). None of the content of this section is novel, all these algorithms can also be found for example in the work of Norell (2007).

2.2.1 Case splitting

A case tree for a function `f` is a tree where each node corresponds to a case split and each leaf corresponds to a clause of `f`.

Example 2.17. Remember the definition of the function `half` from Example 1.3:

$$\begin{aligned} &\texttt{half} : \mathbb{N} \to \mathbb{N} \\ &\texttt{half zero} &&= \texttt{zero} \\ &\texttt{half (suc zero)} &&= \texttt{zero} \\ &\texttt{half (suc (suc } n\texttt{))} &&= \texttt{suc (half } n\texttt{)} \end{aligned} \tag{2.16}$$

It can be represented by a case tree as follows:

$$[(n : \mathbb{N})]\ \underline{n} \begin{cases} [] \ \texttt{zero} \mapsto \texttt{zero} \\ [(m : \mathbb{N})]\ (\texttt{suc}\ \underline{m}) \begin{cases} [] \ (\texttt{suc zero}) \mapsto \texttt{zero} \\ [(k : \mathbb{N})]\ (\texttt{suc (suc } k\texttt{)}) \mapsto \texttt{suc (half } k\texttt{)} \end{cases} \end{cases} \tag{2.17}$$

At each internal node, the variable on which the case split is performed is underlined.

We define case splitting here for simple datatypes; Section 2.2.3 extends it to indexed datatypes.

Definition 2.18 (Pattern specialization, simply-typed version)**.** Let $\bar{p}$ be a list of patterns of type Δ with pattern variables Φ, i.e. $\Phi \vdash \lceil \bar{p} \rceil : \Delta$. If $(x : \mathtt{D}) \in \Phi$ for some datatype $\mathtt{D}$ and $\mathtt{c} : \Delta_{\mathtt{c}} \to \mathtt{D}$ is a constructor of $\mathtt{D}$, then we write $\bar{p} \Rightarrow^{x}_{\mathtt{c}} \bar{p}[x \mapsto \mathtt{c}\ \bar{y}]$ where $\bar{y}$ are fresh variables of type $\Delta_{\mathtt{c}}$.

Definition 2.19 (Case splitting, simply-typed version)**.** Let $\bar{p}$ be a list of well-typed patterns with pattern variables Φ and let $(x : \mathtt{D}) \in \Phi$. A *splitting* (also called a *direct covering*) of $\bar{p}$ is the set of patterns $\bar{p}_i$ such that $\bar{p} \Rightarrow^{x}_{\mathtt{c_i}} \bar{p}_i$ where $\mathtt{c_1}, \ldots, \mathtt{c_n}$ are the constructors of $\mathtt{D}$.

Definition 2.20 (Case tree)**.** A *case tree* for a function $\mathtt{f} : \Delta \to T$ is either an *internal node* or a *leaf node*. Each node has a label of the form $[\Phi]\bar{p}$ where Φ is a telescope and $\bar{p}$ is a list of patterns of type Δ with free variables from the telescope Φ. The root node of a case tree has label $[\Delta]\bar{x}$ where $\bar{x}$ are fresh variables.

- An internal node with label $[\Phi]\bar{p}$ is of the form

$$[\Phi]\bar{p} \begin{cases} ct_1 \\ \ldots \\ ct_n \end{cases} \tag{2.18}$$

  where $ct_1, \ldots, ct_n$ are again case trees with labels $[\Phi_1]\bar{p}_1, \ldots, [\Phi_n]\,\bar{p}_n$ and $\bar{p}_1, \ldots, \bar{p}_n$ form a splitting of $\bar{p}$.

- A leaf node with label $[\Phi]\bar{p}$ is of the form $[\Phi]\bar{p} \mapsto u$ where $\Phi \vdash u : T[\Delta \mapsto \lceil \bar{p} \rceil]$.

Most of the time we don't write down the telescope $[\Phi]$ because it can easily be reconstructed from $\bar{p}$ and Δ.

Example 2.21. An absurd clause in a definition by pattern matching corresponds to an internal node with zero subtrees in the case tree. For example, the case tree for the function `absurd` (Example 1.7) is given as follows:

$$[(A : \mathtt{Set}_\ell)(x : \bot)]\ A; \underline{x}\,\{ \tag{2.19}$$

In this case, we may also write $[(x : \bot)]\ \underline{x} \Rightarrow x$ to make it clear the splitting is empty.

To construct a case tree from a given set of clauses, in each node one pattern variable is chosen on which to split the pattern. This variable must be a *blocking variable*: in at least one of the function clauses, there has to be a constructor pattern in the position of this variable. More precisely, a variable x

in the pattern $\bar{p}$ is blocking if either there is a given clause $f\ \bar{q} = t$ such that $\text{MATCH}(\bar{p}, \lceil \bar{q} \rceil) \Rightarrow \sigma$ and the value assigned to x by σ is a constructor form; or there is a given clause $f\ \bar{q}\ \Cap\ y$ such that $\text{MATCH}(\bar{p}, \lceil \bar{q} \rceil) \Rightarrow \sigma$ and the value assigned to x by σ is equal to the variable y.

When a blocking variable has been found, the pattern is split on that variable and a subtree is constructed for each of the patterns in the splitting. This process is repeated until there are no more blocking variables, at which point the leaf node is filled in by the right-hand side of the corresponding function clause.

Not all definitions by pattern matching can be represented as a case tree. In particular, the patterns could overlap or be otherwise not obtainable by a sequence of case splits.

Example 2.22. Consider a function `minimum` defined as follows:

$$\begin{aligned} &\texttt{minimum} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &\texttt{minimum zero}\ y &&= y \\ &\texttt{minimum}\ x\ \texttt{zero} &&= x \\ &\texttt{minumum}\ (\texttt{suc}\ x)\ (\texttt{suc}\ y) &&= \texttt{suc}\ (\texttt{minimum}\ x\ y) \end{aligned} \tag{2.20}$$

There is no case tree that corresponds precisely to this definition: if the first case split is on the variable x then the second clause will fail to hold definitionally, but if the first case split is on y then the first one will fail to hold.

In cases like this, Agda uses the first-match semantics construct an approximation of clauses by a case tree. However, this can produce unexpected results for the user of the language. For example, the second clause of the definition of `minimum` will fail to hold definitionally. In our previous work, we explored whether it is possible to allow overlapping patterns (Cockx, Devriese, and Piessens, 2014a). In Agda Lite, we avoid the problem by disallowing definitions that cannot be presented as a case tree. The same behaviour can be obtained in the full Agda language by enabling the `--exact-split` option.

2.2.2 Structural recursion

To guarantee termination of functions by pattern matching, they are required to be *structurally recursive*. This means that the arguments of recursive calls should be *structurally smaller* than the pattern on the left-hand side. For functions with multiple arguments, the function should be structurally recursive on one of its arguments, i.e. there should be some k such that $s_k \prec p_k$ for each clause $\texttt{f}\ \bar{p} = t$ and each recursive call $\texttt{f}\ \bar{s}$ in t.

$$\frac{\Gamma \vdash \mathtt{c}\ t_1\ \ldots\ t_n : \mathtt{D}\ \bar{u} \qquad \Gamma \vdash t_i : \Phi \to \mathtt{D}\ \bar{v} \qquad \Gamma \vdash (s_1; \ldots; s_k) : \Phi}{t_i\ s_1\ \ldots\ s_k \prec \mathtt{c}\ t_1\ \ldots\ t_n}$$

$$\frac{s \prec t_i \qquad \Gamma \vdash \mathtt{c}\ t_1\ \ldots\ t_n : \mathtt{D}\ \bar{u} \qquad \Gamma \vdash t_i : \mathtt{D}\ \bar{v}}{s \prec \mathtt{c}\ t_1\ \ldots\ t_n}$$

Figure 2.6: The structural order $\prec$ is used to check termination and to detect cycles during unification. In the first rule, t_i can be any argument $t_1, \ldots, t_n$ which is a recursive argument of the constructor `c`. The additional arguments $s_1, \ldots, s_k$ are required to deal with higher-order constructor arguments and can be ignored for the most of the datatypes in this thesis.

The structural order $\prec$ is defined in Figure 2.6. In both rules, the constructor `c` must be fully applied. Additionally, in both rules t_i is required to be a *recursive* argument of `c`, i.e. the type of the i'th argument of `c` must be of the form $\Phi \to \mathtt{D}\ \bar{v}$, and it must be applied to arguments of type Φ.

This definition is somewhat different from the one given by Abel and Altenkirch (2002) which is used by Goguen et al. (2006). Our definition describes the same relation in case the left- and right-hand sides are elements of the datatype `D`, but it enforces that the left- and right-hand sides are actually elements of the datatype. This prevents odd structural orders that are allowed by the original definition. For example, if we have a datatype `D` with a constructor $\mathtt{c} : (A \to \mathtt{D}) \to \mathtt{D}$, then the definition by Goguen et al. allows us to derive for any $f : A \to \mathtt{D}$ that $f \prec \mathtt{c}\ f \prec \mathtt{c}$, even though f doesn't occur in `c`. While we couldn't find an example where this definition causes an actual problem, using a definition that prevents this kind of spurious orderings helps us in the translation of pattern matching to eliminators (in particular Lemma 4.8, Lemma 4.13 and Lemma 4.18).

It is possible that a function is not structurally recursive but can still be seen to be terminating according to a more complex criterion such as size-change termination (Lee, Jones, and Ben-Amram, 2001). Such more complex termination criteria fall outside the scope of this thesis.

2.2.3 Unification of datatype indices

When doing a case split on a variable of an indexed datatype, the typechecker must decide which constructors can be used to construct a term of a particular type, and under which constraints. For example, consider the inductive family $m \leq n$ with constructors `lz` and `ls` (Example 2.12). To do a case split on a

$$\frac{x \notin FV(v)}{(x \stackrel{?}{=} v)\Theta \xrightarrow{[x \mapsto v]} \Theta[x \mapsto v]}$$

$$\frac{y \notin FV(u)}{(u \stackrel{?}{=} y)\Theta \xrightarrow{[y \mapsto u]} \Theta[y \mapsto u]}$$

$$\frac{\Gamma \vdash u = v : T}{(u \stackrel{?}{=} v)\Theta \overset{[]}{\Rightarrow} \Theta}$$

$$\frac{}{(\mathtt{c}\ \bar{u} \stackrel{?}{=} \mathtt{c}\ \bar{v})\Theta \overset{[]}{\Rightarrow} (\bar{u} \stackrel{?}{=} \bar{v})\Theta}$$

$$\frac{\mathtt{c}_1 \neq \mathtt{c}_2}{(\mathtt{c}_1\ \bar{u} \stackrel{?}{=} \mathtt{c}_2\ \bar{v})\Theta \xrightarrow{\textbf{conflict}} \bot}$$

$$\frac{u \prec v}{(u \stackrel{?}{=} v)\Theta \xrightarrow{\textbf{cycle}} \bot}$$

$$\frac{v \prec u}{(u \stackrel{?}{=} v)\Theta \xrightarrow{\textbf{cycle}} \bot}$$

$$\frac{\Gamma \vdash u = u' : A \qquad \Gamma \vdash v = v' : A \qquad (u \stackrel{?}{=} v)\Theta \overset{\sigma}{\Rightarrow} \Theta'}{(u' \stackrel{?}{=} v')\Theta \overset{\sigma}{\Rightarrow} \Theta'}$$

Figure 2.7: These unification transitions are used for case splitting on a variable of an indexed datatype.

variable of type $n \leq \mathtt{zero}$ as in the definition of `antisym` (Example 1.12), the typechecker has to decide for what arguments the two constructors produce a result of the form $n \leq \mathtt{zero}$. The constructor `lz` can only be used at this type in case $n = \mathtt{zero}$, and `ls` cannot be used at all because $\mathtt{suc}\ n$ can never be equal to `zero`.

In general, to check whether a constructor $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{v}$ can be used at type $\mathtt{D}\ \bar{u}$, the typechecker has to *unify* $\bar{u}$ with $\bar{v}$. It constructs the unification problem Θ, where Θ is a list of equations $\bar{u} \stackrel{?}{=} \bar{v}$, and applies the following unification transitions to simplify the problem step by step (Figure 2.7):

$$\frac{}{() \overset{[]}{\Rightarrow}{}^{*} ()}$$

$$\frac{\Theta \overset{\tau}{\Rightarrow} \Theta' \qquad \Theta' \overset{\sigma}{\Rightarrow}{}^{*} ()}{\Theta \overset{\tau;\sigma}{\Longrightarrow}{}^{*} ()}$$

$$\frac{\Theta \overset{\textbf{conflict/cycle}}{\dashrightarrow} \bot}{\Theta \Rightarrow^{*} \bot}$$

$$\frac{\Theta \overset{\tau}{\Rightarrow} \Theta' \qquad \Theta' \Rightarrow^{*} \bot}{\Theta \Rightarrow^{*} \bot}$$

Figure 2.8: The unification algorithm applies the unification transitions of Figure 2.7 until it finds a most general unifier, it ends in a conflict or cycle, or no more transition applies.

Solution. The solution rule (also called the *substitution rule* or the *coalescence rule*) solves an equation $x \overset{?}{=} v$ if one side is a variable. This rule cannot be applied if x occurs free in v.

Deletion. The deletion rule removes equations where the left- and right-hand side are definitionally equal.

Injectivity. The injectivity rule simplifies equations of the form $\mathtt{c}\ x_1\ \ldots\ x_n \overset{?}{=} \mathtt{c}\ y_1\ \ldots\ y_n$ if $\mathtt{c}$ is a constructor. For example, this rule allows us to simplify $\mathtt{suc}\ m \overset{?}{=} \mathtt{suc}\ n$ to $m \overset{?}{=} n$.

Conflict. The conflict rule refutes absurd equations of the form $\mathtt{c}_1\ x_1\ \ldots\ x_m \overset{?}{=} \mathtt{c}_2\ y_1\ \ldots\ y_n$ where $\mathtt{c}_1$ and $\mathtt{c}_2$ are two distinct constructors. For example, it allows us to conclude that an equation of the form $\mathtt{zero} \overset{?}{=} \mathtt{suc}\ y$ is absurd.

Cycle. The cycle refutes detects cyclical equations of the form $x \overset{?}{=} \mathtt{c}\ y_1\ \ldots\ y_n$ where x occurs rigidly in $y_1, \ldots, y_n$. For example, it allows us to detect that an equation of the form $n \overset{?}{=} \mathtt{suc}\ n$ is absurd. To formalize this rule, we use the structural order $\prec$ (Figure 2.6).

Exhaustively applying these rules whenever they are applicable terminates by the usual argument, with three possible outcomes (Jouannaud and Kirchner, 1990):

Positive success: All equations have been solved, yielding a most general unifier σ. In this case, we write $\Theta \overset{\sigma}{\Rightarrow}^* ()$.

Negative success: Either the **conflict** or the **cycle** rule applies, meaning that there exist no unifiers, i.e. this case is absurd. In this case, we write $\Theta \Rightarrow^* \bot$.

Failure: An equation is reached for which no transition applies, meaning that the problem is too hard to be solved by this unification algorithm.

The third outcome can occur for example when the unification algorithm encounters an equation between two functions. Nevertheless, the algorithm is complete for *constructor forms*: if both $\bar{u}$ and $\bar{v}$ are built from constructors and variables only, then unification never results in a failure.

In the next chapter, we describe unification in a dependently typed setting in general and give an extended and *proof-relevant* version of the algorithm presented here. Now we extend the definitions of pattern specialization and case splitting to indexed datatypes.

Definition 2.23 (Pattern specialization, general version)**.** Let $\bar{p}$ be a list of patterns of type Δ with pattern variables Φ and let $(x : \mathtt{D}\ \bar{u}) \in \Phi$ for some datatype $\mathtt{D}$ and $\mathtt{c} : \Delta_{\mathtt{c}} \to \mathtt{D}\ \bar{v}$ be a constructor of $\mathtt{D}$. If $(\bar{u} \overset{?}{=} \bar{v}) \overset{\sigma}{\Rightarrow}^* ()$, then we write $\bar{p} \Rightarrow^x_{\mathtt{c}} \bar{p}\sigma'[x \mapsto \mathtt{c}\ (\Delta_{\mathtt{c}}\sigma')]$ where σ' substitutes a pattern variable z for an inaccessible pattern $.(z\sigma)$ for any z in the domain of σ. On the other hand, if $(\bar{u} \overset{?}{=} \bar{v}) \Rightarrow^* \bot$ then we write $\bar{p} \Rightarrow^x_{\mathtt{c}} \bot$.

Example 2.24. Let $\Delta = (n : \mathbb{N})(x : \mathtt{Vec}\ A\ n)$ and $\bar{p} = (n; x)$ and consider the constructor $\mathtt{nil} : \mathtt{Vec}\ A\ \mathtt{zero}$. We have $(n \overset{?}{=} \mathtt{zero}) \xrightarrow{[n \mapsto \mathtt{zero}]} ()$, so $(n; x) \Rightarrow^x_{\mathtt{nil}} (.\mathtt{zero}; \mathtt{nil})$.

Definition 2.25 (Case splitting, general version)**.** Let $\bar{p}$ be a list of well-typed patterns with pattern variables Φ and let $(x : \mathtt{D}\ \bar{u}) \in \Phi$. Suppose that for every constructor $\mathtt{c_i} : \Delta_i \to \mathtt{D}\ \bar{v}_i$ we either have $\bar{p} \Rightarrow^x_{\mathtt{c_i}} \bar{p}_i$ or $\bar{p} \Rightarrow^x_{\mathtt{c_i}} \bot$. Then a *splitting* (also called a *direct covering*) of $\bar{p}$ is the set of patterns $\bar{p}_i$ such that $\bar{p} \Rightarrow^x_{\mathtt{c_i}} \bar{p}_i$.

It is possible for a case split to be empty, even if the datatype has more than zero constructors. In this case it is possible to have internal node of the case tree with no subtrees, this corresponds to an absurd clause of the function.

The definition of a valid case tree remains unchanged from the simply-typed version, provided we use the new definitions of pattern specialization and case splitting. Now we define what we mean by a valid definition by pattern matching.

Definition 2.26 (Pattern matching validity)**.** A definition by pattern matching is valid if the following conditions are satisfied:

- The (non-absurd) clauses of the definition arise as the leaf nodes of a case tree for `f`.
- The absurd clauses of the definition arise as the internal nodes of the same case tree for which there are no subtrees.
- There is an index i such that the clauses of `f` are structurally recursive on their ith argument.

Example 2.27. We build a case tree for the function `antisym` (Example 1.12). The case tree is given in Figure 2.9. The first case split on the argument of type $m \leq n$ results in two cases: either `lz` n or `ls` $k\ l\ x$. The second case split on the argument of type $n \leq$ `zero` is more interesting:

- For the `lz` constructor, unification tells us that n must be equal to `zero`:

$$\begin{array}{r} n \stackrel{?}{=} \texttt{zero}, \\ \texttt{zero} \stackrel{?}{=} k \end{array} \xrightarrow{[n \mapsto \texttt{zero}]} \texttt{zero} \stackrel{?}{=} k \xrightarrow{[k \mapsto \texttt{zero}]} () \tag{2.21}$$

  We renamed the argument n of `lz` to k to avoid a name conflict with the argument n of `antisym`.

- For the `ls` constructor, unification ends in a conflict, as `zero` cannot be equal to something of the form `suc` k :

$$\begin{array}{r} n \stackrel{?}{=} \texttt{suc}\ k, \\ \texttt{zero} \stackrel{?}{=} \texttt{suc}\ l \end{array} \xrightarrow{[n \mapsto \texttt{suc}\ k]} \texttt{zero} \stackrel{?}{=} \texttt{suc}\ l \xrightarrow{\textbf{conflict}} \bot \tag{2.22}$$

  As in the previous case, we renamed the arguments m and n of `ls` to k and l.

This is reflected in the upper subtree of the case tree, where there is only a case for the constructor `lz`, and none for `ls`. Similarly, the second subtree only has a case for the `ls` constructor, not for `lz`.

2.3 Pattern matching without K

In the previous sections, we have seen how to define functions in Agda Lite by dependent pattern matching. If we check that all definitions by pattern

$$
m\ n\ \underline{x}\ y \begin{cases} .\mathtt{zero}\ n\ (\mathtt{lz}\ .n)\ \underline{y} \\ \quad \{\ .\mathtt{zero}\ .\mathtt{zero}\ (\mathtt{lz}\ .\mathtt{zero})\ (\mathtt{lz}\ .\mathtt{zero}) \mapsto \mathtt{refl} \\ .(\mathtt{suc}\ k)\ .(\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ x)\ \underline{y} \\ \quad \begin{cases} .(\mathtt{suc}\ k)\ .(\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ \underline{l}\ x)\ (\mathtt{ls}\ .l\ .k\ y) \\ \qquad \mapsto \mathtt{cong}\ \mathtt{suc}\ (\mathtt{antisym}\ k\ l\ x\ y) \end{cases} \end{cases}
$$

Figure 2.9: A representation of the function `antisym` (Example 1.12) by a case tree. While there are two subtrees for the case split on x, each split on y only has a single subcase due to the constraints on the type of y.

matching are valid (or let a typechecker do it for us), then we can trust that the language satisfies canonicity, strong normalization, and the Church-Rosser property. Together, these properties imply that our language is consistent, i.e. that there can be no term of type $\bot$ in the empty context. However, this is no longer true once we add extra axioms such as univalence. In particular, we can prove K by pattern matching (1.29), which together with univalence implies an inconsistency.

It is exactly this K rule that is responsible for the incompatibility between univalence and dependent pattern matching. This was shown by Goguen et al. (2006): definitions by pattern matching can be translated to ones that only use the standard datatype eliminators and the K rule. Intuitively, this makes sense because HoTT (and univalence in particular) allows us to encode important information in equality proofs, while K is exactly the assertion that there is no such information.

One of our main goals is to formulate a version of dependent pattern matching that doesn't have these problems and can hence be used in many more settings, in particular in HoTT. For this, we need to use a different unification algorithm in the definition of pattern specialization (Definition 2.23). This new unification algorithm is the subject of the next chapter. In addition, we need to make a small change to the definition of structural recursion.

In this section, we present three additional restrictions to definitions by dependent pattern matching that, when used together, guarantee that it is impossible to prove K or any of its consequences. This allowing pattern matching to be used together with axioms from HoTT (Section 2.3.1). We also discuss the implementation of our criterion in Agda and compare it to the old criterion offered by Agda (Section 2.3.3).

The first two of these three restrictions are meant merely to ensure that the untyped unification algorithm from this chapter computes a conservative approximation of the more expressive, typed unification algorithm in the next

chapter. However, we think it is useful to first see how the existing definitions have to be restricted in order to remove the dependency on K before extending them in other ways. In addition, these restrictions may be useful for someone who wants to implement dependent pattern matching but does not want to deal with the complexity of a typed unification algorithm.

2.3.1 Three restrictions for avoiding K

For function definitions that match only on simple types, like the function `half` (Example 1.3), each case split corresponds exactly to one application of the standard eliminator for $\mathbb{N}$, hence the K rule is not needed. However, the unification algorithm used for case splitting on an inductive family depends crucially on the K rule, so it has to be restricted to remove this dependence.

The first restriction

As the first and most important restriction, we prohibit the use of the **deletion** step of the unification algorithm. According to this restriction, the definition of K by pattern matching (1.29) is not allowed, as case splitting on the argument of type $a \equiv_A a$ produces a unification problem $a \stackrel{?}{=} a$, which fails without the **deletion** step of the unification algorithm. In contrast, the definition of J (1.27) is still allowed since it only uses the **solution** rule. Likewise, the definitions of `sym`, `trans`, `cong`, `subst` (Example 1.13), and `antisym` (Example 1.12) in the introduction are also still accepted.

If unification still results in a success (a positive or negative one) then the original rules would have given the same result. Where the original algorithm was complete for constructor forms, the version without the **deletion** rule is only complete for *linear* constructor forms (i.e. ones where each variable occurs only once).

The second restriction

Example 2.28. As an example of why a further restriction is needed, consider the following weaker variant of K:

$$\begin{aligned} &\texttt{weakK} \;:\; (P : \texttt{refl} \equiv_{a \equiv_A a} \texttt{refl} \to \texttt{Set}) \to \\ &\qquad\quad (p : P\ \texttt{refl})(e : \texttt{refl} \equiv_{a \equiv_A a} \texttt{refl}) \to P\ e \\ &\texttt{weakK}\ P\ p\ \texttt{refl} = p \end{aligned} \tag{2.23}$$

The type of `weakK` says that any proof e of $\texttt{refl} \equiv_{a \equiv_A a} \texttt{refl}$ is equal to `refl`. In the HoTT interpretation of identity proofs as paths, this would mean that there are no non-trivial paths between paths, i.e. all spaces have a dimension of at most 1. Like the regular K, this `weakK` does not follow from the standard rules of type theory and is incompatible with univalence (Kraus and Sattler, 2015).

The second restriction limits the **injectivity** rule to be applicable only to certain constructors, excluding for example `refl`. To formulate it, we first need to introduce some additional terminology.

Definition 2.29 (Forced constructor argument, rigid occurrence)**.** A variable x occurs *rigidly* in a term t if either $t = x$ or t is of the form $\texttt{c}\ t_1\ \ldots\ t_n$ where x occurs rigidly in one of the t_i and t_i is not a forced argument of `c`.

An argument of a constructor $\texttt{c} : \Delta \to \texttt{D}\ \bar{u}$ is *forced* if it occurs rigidly in one of the indices $\bar{u}$.

As an example, the argument $(n : \mathbb{N})$ is a forced argument of $\texttt{cons} : (n : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ n) \to \texttt{Vec}\ A\ (\texttt{suc}\ n)$. The concept of a forced constructor argument was introduced by Brady, McBride, and McKinna (2003) for efficiently compiling dependently typed programs. We use it here to describe when it is safe to apply the **injectivity** rule.

Definition 2.30 (Invertible constructor)**.** A constructor $\texttt{c} : \Delta \to \texttt{D}\ \bar{u}$ is *invertible* if the indices $\bar{u}$ consist only of invertible constructors and variables bound in Δ, and no variable from Δ occurs more than once in a non-forced position in $\bar{u}$.

In particular, constructors of non-indexed datatypes such as $\mathbb{N}$ are always invertible. Likewise, the constructor $\texttt{cons} : (n : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ n) \to \texttt{Vec}\ A\ (\texttt{suc}\ n)$ is invertible. In contrast, $\texttt{refl} : u \equiv_A u$ is not an invertible constructor unless u consists itself completely of invertible constructors: $\texttt{refl} : \texttt{zero} \equiv_{\mathbb{N}} \texttt{zero}$ is invertible, but $\texttt{refl} : n \equiv_{\mathbb{N}} n$ for variable $n : \mathbb{N}$ is not.

The second restriction to the unification algorithm is specified as follows: when applying the **injectivity** step on the equation $\texttt{c}\ \bar{s} \stackrel{?}{=} \texttt{c}\ \bar{t}$, the constructor `c` has to be invertible. This prevents the definition of `weakK` because the constructor $\texttt{refl} : a \equiv_A a$ is not invertible. This restriction is a conservative approximation of the more general principle of *higher-dimensional unification*, as described in the next chapter (Section 3.4).

With the definition of forced constructor arguments, we can also relax the first restriction a bit. In particular, when applying **injectivity** to an equation

$\mathtt{c}\ \bar{s} \stackrel{?}{=} \mathtt{c}\ \bar{t}$, it is safe to skip unification of the forced arguments of $\mathtt{c}$. This allows us to avoid some situations where unification would otherwise require the **deletion** rule.

Example 2.31. Let $\mathtt{Fin}\ n$ be the type of natural numbers strictly less than n. It can be defined as an indexed datatype as follows:

$$\begin{aligned}&\textbf{data}\ \mathtt{Fin} : \mathbb{N} \to \mathtt{Set}\ \textbf{where}\\ &\quad \mathtt{fzero} \ : (n : \mathbb{N}) \to \mathtt{Fin}\ (\mathtt{suc}\ n)\\ &\quad \mathtt{fsuc} \ : (n : \mathbb{N}) \to \mathtt{Fin}\ n \to \mathtt{Fin}\ (\mathtt{suc}\ n)\end{aligned} \tag{2.24}$$

Consider the $\leq_{\mathtt{Fin}}$ relation on $\mathtt{Fin}$:

$$\begin{aligned}&\textbf{data}\ \_ \leq_{\mathtt{Fin}} \_ : (n : \mathbb{N}) \to \mathtt{Fin}\ n \to \mathtt{Fin}\ n \to \mathtt{Set}\ \textbf{where}\\ &\quad \mathtt{lz} : (n : \mathbb{N})(y : \mathtt{Fin}\ (\mathtt{suc}\ n)) \to \mathtt{fzero}\ n \leq_{\mathtt{Fin}} y\\ &\quad \mathtt{ls} : (n : \mathbb{N})(x\ y : \mathtt{Fin}\ n) \to x \leq_{\mathtt{Fin}} y \to \mathtt{fsuc}\ n\ x \leq_{\mathtt{Fin}} \mathtt{fsuc}\ n\ y\end{aligned} \tag{2.25}$$

When case splitting on an argument of type $i \leq_{\mathtt{Fin}} i$ where $i : \mathtt{Fin}\ n$, the unification algorithm applies **injectivity** to the equation $\mathtt{fsuc}\ n\ x \stackrel{?}{=} \mathtt{fsuc}\ n\ y$. Since the first argument of the constructor $\mathtt{fsuc}$ is forced, the corresponding equation $n \stackrel{?}{=} n$ can be skipped. Otherwise, the **deletion** rule would be needed to solve this equation.

The third restriction

One important but easily overlooked detail in the translation of dependent pattern matching to eliminators by Goguen et al. (2006) is that the type of the argument on which the function is structurally recursive must be a datatype. When working in a theory without the K rule, this restriction becomes very important.

Example 2.32. When we allow recursion on an argument of variable type, we can refute univalence.[2] Let $\mathtt{One} : \mathtt{Set}$ be a datatype with a single constructor $\mathtt{wrap} : (\bot \to \mathtt{One}) \to \mathtt{One}$. Since $\mathtt{wrap}$ is a constructor and hence injective, it gives rise to an equivalence between $\mathtt{One}$ and $\bot \to \mathtt{One}$. By univalence, we have a proof $\mathtt{iso} = \mathtt{ua}\ \mathtt{wrap} : \mathtt{One} \equiv_{\mathtt{Set}} (\bot \to \mathtt{One})$. Now we define a function $\mathtt{noo}$ by pattern matching as follows:

$$\begin{aligned}&\mathtt{noo} : (X : \mathtt{Set}) \to (\mathtt{One} \equiv_{\mathtt{Set}} X) \to X \to \bot\\ &\mathtt{noo}\ .\mathtt{One}\ \mathtt{refl}\ (\mathtt{wrap}\ f) = \mathtt{noo}\ (\mathtt{One} \to \bot)\ \mathtt{iso}\ f\end{aligned} \tag{2.26}$$

[2] This example has been adapted from the ones given by Maxime Dénès and Conor McBride on the Agda mailing list (see `https://lists.chalmers.se/pipermail/agda/2014/006252.html`).

First, `noo` pattern matches on the proof of $\texttt{One} \equiv_{\texttt{Set}} X$, forcing X to be equal to `One`. Next, it proceeds by induction on its third argument, which first had type X but is now of type `One`.

Once we have `noo`, we can prove $\bot$ as follows:

$$\begin{aligned} &\texttt{false} : \bot \\ &\texttt{false} = \texttt{noo}\ (\texttt{One} \to \bot)\ \texttt{iso}\ (\texttt{absurd}\ \texttt{One}) \end{aligned} \tag{2.27}$$

According to the naive interpretation, the function `noo` is structurally recursive on its third argument. However, the type of this argument changes from `One` in the argument position to $\bot \to \texttt{One}$ in the recursive call. In effect, the definition of `noo` first strips the `wrap` constructor from its third argument to fool the termination checker, only to apply it again via a backdoor using the equality `iso`.

The kind of recursion used in the above example is not allowed by the eliminator for the `One` type, and is in fact incompatible with univalence as made evident by the proof `false` of $\bot$. So we need to be careful to disallow this kind of recursion, both in the proof and in the implementation of the criterion. In general, the type of the argument on which the function is structurally recursive must be of the form $\texttt{D}\ \bar{u}$ where `D` is a datatype.

2.3.2 Soundness of the criterion

These three restrictions rule out a direct definition of `K` (1.29) or a weaker form of it (2.23), as well as other undesired definitions such as `noo` (Example 2.32). To be certain that this is *enough*, we prove that any definition by pattern matching satisfying this criterion could as well be written in a simpler language without dependent pattern matching but only datatype eliminators, like Goguen et al. (2006) did for pattern matching *with* K. In addition, we also prove that the definition by eliminators still has the same computation rules as the one by pattern matching. Formally, we prove the following:

Corollary 2.33. *Let* $\texttt{f} : (\bar{t} : \Delta) \to T$ *be a function given by a valid case tree, adhering to the following three restrictions:*

- *When case splitting on a variable of an indexed datatype, it is not allowed to use the* ***deletion*** *step of the unification algorithm, except when unifying forced constructor arguments.*
- *When applying the* ***injectivity*** *step on the equation* $\texttt{c}\ \bar{s} \stackrel{?}{=} \texttt{c}\ \bar{t}$*, the constructor* `c` *has to be invertible.*

- *When checking termination of a function by pattern matching, the type of the argument on which the function is structurally recursive must be of the form* `D` $\bar{u}$ *where* `D` *is a datatype.*

Then we can construct a term `f'` $: (\bar{t} : \Delta) \to T$ *constructed from eliminators only. Moreover, define* $\{e\}^{\mathtt{f} \mapsto \mathtt{f'}}$ *by replacing all occurrences of* `f` *by* `f'` *in* e*. Then* `f'` *satisfies* `f'` $\bar{t} = \{u\}^{\mathtt{f} \mapsto \mathtt{f'}}$ *whenever* `f` $\bar{t} = u$*, i.e. it has the same reduction behaviour as* `f`*.*

Proof. This is a special case of the results in Chapter 4, in particular Theorem 4.27 and Theorem 4.29. To show that the injectivity rule for indexed datatypes as given in this chapter is valid, we also need Corollary 3.56. □

This result gives us more than just the assurance that we cannot prove anything we couldn't prove with just the basic rules of type theory: it also gives us an effective method to compile definitions by pattern matching to datatype eliminators. Agda currently only guarantees that this compilation could be performed in theory, but the Equations package for Coq by Sozeau (2010) takes this idea into practice by actually performing the translation.

2.3.3 Comparison with the old criterion

We implemented the criterion for pattern matching without K as an optional flag for Agda 2.4.0 and later called `--without-K`, replacing an older version of `--without-K`. In older versions of Agda, it attempted to detect definitions by pattern matching that make use of the K rule by means of a syntactic check. It is described as follows by Norell, Abel, and Danielsson (2012):

> If the flag is activated, then Agda only accepts certain case-splits. If the type of the variable to be split is `D pars ixs`, where `D` is a data (or record) type, `pars` stands for the parameters, and `ixs` the indices, then the following requirements must be satisfied:
>
> - The indices `ixs` must be applications of constructors (or literals) to distinct variables. Constructors are usually not applied to parameters, but for the purposes of this check constructor parameters are treated as other arguments.
> - These distinct variables must not be free in `pars`.

This old criterion has been criticized many times, for being too restrictive (Sicard-Ramírez, 2013), for having unclear semantics (Reed, 2013), and for

permitting definitions that are not equivalent to ones that only use datatype eliminators (Altenkirch, 2012; Cockx, 2014). These errors allowed one to prove (weaker versions of) the K rule, such as Example 2.28. One reason to prefer our criterion is that we can prove it doesn't contain such errors (Section 4.3). However, we should also compare the generality of the two criteria, i.e. what kind of definitions are still allowed by each.

On the one hand, the old criterion implies that the **deletion** rule is never used during unification. It guarantees that all unification problems generated by pattern matching are of the form $\bar{u} \stackrel{?}{=} \bar{v}_i$ where $\bar{u}$ consists of constructors applied to free variables and each variable occurs only once in $\bar{u}$. Moreover, since new constructors introduced by case splitting are applied to fresh variables, the variables in $\bar{u}$ are not free in $\bar{v}_i$. Both the **solution** and the **injectivity** step preserve these three properties, hence unification never reaches an equation of the form $x \stackrel{?}{=} x$.

On the other hand, the old criterion does not imply that the second or third restrictions were satisfied. But this is actually a defect, allowing one to prove a weaker version of the K rule (Cockx, 2014), similar to Example 2.28. Likewise, the old criterion fails to forbid the definition of `noo` (Example 2.32). So the fact that our criterion is more restrictive in this case is actually a good thing.

Apart from these two extra necessary restrictions, our criterion is in fact strictly more general than the old one. For example, the old criterion allows us to pattern match with `refl` on an argument of type $k + l \equiv_{\mathbb{N}} m$ (where $k, l, m : \mathbb{N}$ are previous arguments), but not on an argument of type $m \equiv_{\mathbb{N}} k + l$. This asymmetry is created by a technical detail in the standard definition of propositional equality as an inductive family: the first argument is a parameter (so it can be anything), while the second one is an index (so it must consist of constructors applied to free variables). In contrast, our criterion allows both variants because we look at the unifications that are performed instead of syntactical artefacts like the distinction between a parameter and an index. Similarly, the old criterion does not allow us to pattern match on an argument of type $n \leq n$ because the variable n occurs twice, while it is allowed by our criterion.

Another advantage of our criterion is that it does not put any requirements on the datatype parameters. This is useful when injectivity is needed for a constructor of a parametrized datatype. For example, the old criterion does not allow case splitting on an argument of type $\texttt{cons}\ x\ xs \equiv_{\texttt{List}\ a} \texttt{cons}\ y\ ys$ where $\texttt{cons} : A \to \texttt{List}\ A \to \texttt{List}\ A$ is the list constructor, since the type A of x and y is a parameter and the constructor `cons` is considered to be applied to this parameter. Our criterion has no such problems. This is especially useful in Agda since module parameters are also considered to be parameters of the

datatypes defined inside that module (Norell, 2007, chapter 4). So with the old criterion, moving a definition to another module can cause an error, but with our criterion this is no longer the case.

The criterion presented here is also more general than the one presented in the previously published versions of this work (Cockx et al., 2014b, 2016a). For example, Example 2.31 was rejected both by the old implementation of `--without-K` and by the previous version of our criterion, but is now accepted.

One limitation of our criterion arises when an equation cannot be solved right away, and is instead postponed until later. As the types of later equations may depend on the solution of these postponed equations, this may cause the types of both sides of an equation to be different. In the next chapter, we extend the unification algorithm so it can deal with such postponed equations.

2.4 Related work

Most implementations of dependent pattern matching in the style of Coquand (1992) assume K. Examples include Agda (when `--without-K` is not enabled), Epigram (McBride and McKinna, 2004; McBride, 2005), Idris (Brady, 2013), the pattern matching construct for Coq described by Barras, Corbineau, Grégoire, Herbelin, and Sacchini (2009), and the Equations package for Coq described by Sozeau (2010).

In an unpublished paper, McBride (1998a) used homogeneous equality and observes like us that the innocent-looking **deletion** rule turns into the rather less innocent K. However, the published version of this work uses the heterogeneous equality, thus making it rely on K. This resulted in a significant simplification by avoiding dependency on equality proofs. In our current work, this extra complexity becomes a feature.

Coq also support a more primitive notion of pattern matching via the `match` construct in Gallina (The Coq development team, 2016). The full version of this construct is

$$\begin{array}{l}\texttt{match}\ e\ \texttt{as}\ x\ \texttt{in}\ D\ \bar{u}\ \texttt{return}\ P\ \texttt{with}\\ \quad \mid c_1\ \bar{y}_1 \Rightarrow e_1\\ \quad \mid \ldots\\ \quad \mid c_n\ \bar{y}_n \Rightarrow e_n\\ \texttt{end}\end{array} \tag{2.28}$$

Coq also allows skipping the parts labelled by `as`, `in`, and `return`, in which case it attempts to construct the motive P automatically. The motive P must

be fully generalized over the indices $\bar{u}$, ensuring that no unification is necessary. Hence this kind of matching also prevents us from proving K. However, it is closer to datatype eliminators than it is to the kind of pattern matching described in this thesis, because it requires the user to give each case split explicitly, and does not perform any unification.

Forcing is a compiler optimization for dependently typed programming languages that erases constructor arguments that are fully determined by the type of the constructor (Brady et al., 2003). For example, the argument $(n : \mathbb{N})$ of the `cons` constructor of `Vec` can be erased because it is fully determined by the type of the constructor. During unification, it is intuitively clear that forced arguments can safely be skipped. This intuition is justified by higher-dimensional unification (Section 3.4). In particular, Corollary 3.56 tells us exactly that forced constructor arguments can be skipped during unification. So higher-dimensional unification justifies the heuristic that 'forced arguments need not be unified'. As far as we know, is this the first formalization of forcing for the purpose of unification.

Lean is a new dependently typed language that also supports dependently typed pattern matching by a translation to eliminators (de Moura et al., 2015). Lean can be used with two instantiations of its core theory: one based on the Calculus of Inductive Constructions that allows proving K, and a second one based on homotopy type theory that doesn't. So using this second instantiation also allows one to use dependent pattern matching without relying on K. The authors cite our work (Cockx et al., 2014b) as an important source of inspiration for the implementation of the Lean system.

Recently, a new version of the Equations package for Coq has been developed that also supports pattern matching without assuming K (Mangin and Sozeau, 2017), based on our earlier work (Cockx et al., 2014b). Similarly to our translation in Chapter 4, it uses a generalization of homogeneous telescopic equality to achieve compilation of pattern matching definitions without K. In contrast to the implementation of our criterion in Agda, the Equations package also performs the actual translation of definitions by pattern matching to eliminators. It also allows the use of the **deletion** rule in some cases that are non-dependent or justified by user-provided instances of K.

In his thesis, Boutillier (2014) describes an algorithm for compiling definitions by pattern matching to eliminators in Coq. The criterion he uses is similar to the old criterion used by Agda: to perform a case distinction on a variable of an inductive family, the indices need to be constructors applied to distinct variables, and those variables must not occur in the parameters. To this, he adds a preprocessing step where indices are erased if they are not used in the return type or if they are determined by the type of the other indices. For the

translation, he constructs a *diagonalizer* based on the skeleton of the indices, encoding the induction principle for a particular subset of the inductive family. Compared to our work, Boutillier doesn't give a closed criterion for when a definition by pattern matching is acceptable in a theory without K. Instead, he provides a desugaring of which the result still has to be checked by Coq. In our opinion, this is bad practice because it requires the user to be aware of the desugaring to predict whether a definition will be accepted. In contrast, our criterion only requires the user to know the unification algorithm to predict its behaviour. The computational behaviour of the desugaring also seems more like an afterthought in Boutillier's work, while it is an essential part of ours. Nevertheless, by analysing whether an argument is actually used, either in the type of a later argument or in the return type, he manages to allow some definitions that are rejected by our criterion. This gives a good heuristic for preprocessing pattern matching definitions to remove superfluous uses of K, so it might be used complementary to our criterion.

Chapter 3

Proof-relevant unification

> *Humankind cannot gain anything without first giving something in return. To obtain, something of equal value must be lost. That is Alchemy's First Law of Equivalent Exchange.*
>
> — Hiromu Arakawa (2001)

Unification is a generic method for solving symbolic equations algorithmically. It is a fundamental algorithm used in many areas in computer science, such as logic programming, type inference, term rewriting, automated theorem proving, and natural language processing. In particular, it is used to check definitions by dependent pattern matching, as we saw in the previous chapter.

In a language that has dependent pattern matching as a primitive (such as Agda), the particularities of the unification rules used become crucial for the language's notion of equality. Indeed, we can match on a proof of $u \equiv_A v$ with the constructor `refl` precisely when the unification algorithm is able to unify u with v. For example, if the unification algorithm uses the deletion rule to remove reflexive equations, then this allows us to prove K by pattern matching. This chapter contains some additional examples of what goes wrong if we naively add postponing to the unification algorithm.

Because of the problems with unification, in the previous chapter we required a number of ad hoc restrictions to preserve soundness, such as removing the deletion rule if the theory doesn't support K. We also saw that it is unsound in general to apply the injectivity rule for constructors of indexed datatypes: it is necessary to restrict it to certain constructors. However, these ad hoc

restrictions make the unification algorithm hard to prove correct, modify, or extend.

The goal of this chapter is to give a *fully typed* account of unification in dependent type theory, in order to solve the problems with untyped unification in general and put unification in type theory back on a solid theoretical foundation. We do this by treating unification problems and unification rules as *internal* to the type theory, rather than belonging to some external tool. This ensures that we don't make use of unspecified assumptions such as uniqueness of identity proofs or injectivity of type constructors.

First, we represent unification problems as a *telescope*, a list of types where each type can depend on values of the previous types. Each type in this telescope corresponds to one equation of the unification problem, and the dependencies reflect the fact that the type of each equation can depend on the solutions of previous equations. This allows us to keep track of the dependencies between the equations precisely.

Secondly, we represent unification rules as *equivalences* between two telescopes of equations. For example, the injectivity rule for the constructor $\texttt{suc} : \mathbb{N} \to \mathbb{N}$ is represented by an equivalence between the equations $\texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n$ and $m \equiv_{\mathbb{N}} n$. This equivalence contains not only the substitution needed to go from one set of equations to the next, but also *evidence* that the unification rule is valid. This gives us a new formal criterion for the correctness of a unification rule in a dependently typed setting.

Finally, we give a novel characterization of the most general unifier as an equivalence between the original telescope of equations and the trivial one. It can be constructed by simply composing the individual unification rules that are used. This definition turns out to be a stronger requirement than the standard definition of a most general unifier, yet it is still satisfied by all unification rules that are used in Agda. The fact that most general unifiers correspond to equivalences not only gives us a very elegant way to define them, but it also turns out to be useful the translation of definitions by dependent pattern matching to eliminators Goguen et al. (2006); Cockx et al. (2016b). This makes it also suitable for languages with a core calculus like Coq (The Coq development team, 2016), Epigram (McBride, 2005), Lean (de Moura et al., 2015), or (potentially) a future version of Agda.

We start this chapter with a general description of our framework for reasoning about unification in a dependently typed setting (Section 3.1). We phrase the basic unification rules in this framework and show how our algorithm can easily be extended by adding more unification rules (Section 3.2). We pay special attention to the computational behaviour of unification rules when viewed as

terms of the type theory (Section 3.3). To augment the power of the unification rules for indexed datatypes, we introduce a new technique called higher-dimensional unification (Section 3.4). We finish the chapter with a discussion of the implementation of our unification algorithm in Agda (Section 3.5).

3.1 Unification in dependent type theory

When dependently typed terms become the subject of unification, it is possible that the unification algorithm encounters *heterogeneous equations*: equations in which the left- and right-hand side have different types, that only become equal after previous equations have been solved. For example, consider the type $\Sigma_{A:\mathtt{Set}} A$ with elements (A, a) packing a type A together with an element a of that type. By injectivity of the pair constructor $\_,\_$, an equation $(A, a) = (B, b)$ can be simplified to $A = B$ and $a = b$, but the type of the second equation is now heterogeneous since $a : A$ and $b : B$. Because traditional unification algorithms only look at the syntax of the terms they are trying to unify, they cause problems when applied to heterogeneous equalities.

Example 3.1. Consider the equation $(\mathtt{Bool}, \mathtt{true}) = (\mathtt{Bool}, \mathtt{false})$ of type $\Sigma_{A:\mathtt{Set}} A$. By injectivity of the constructor $\_,\_$, we can simplify this equation to the two equations $\mathtt{Bool} = \mathtt{Bool}$ and $\mathtt{true} = \mathtt{false}$. If we postpone the first equation, we can derive an absurdity from the second equation. However, this line of reasoning depends on the principle of *equality of second projections*, which is equivalent to K (Streicher, 1993). Indeed, if we have access to univalence then we can prove that $(\mathtt{Bool}, \mathtt{true})$ can be identified with $(\mathtt{Bool}, \mathtt{false})$ of type $\Sigma_{A:\mathtt{Set}} A$, so allowing the injectivity rule in this case is incompatible with univalence.

On the other hand, consider the exact same unification problem $(\mathtt{Bool}, \mathtt{true}) = (\mathtt{Bool}, \mathtt{false})$, but this time the type of the equation is a non-dependent product $\mathtt{Set} \times \mathtt{Bool}$. In this case it *is* possible to derive an absurdity, even in a theory with univalence. However, a syntax-directed unification algorithm can never distinguish between these two cases.

Example 3.2. Suppose we define two copies $\mathtt{Bool}_1$ and $\mathtt{Bool}_2$ of the boolean type with constructors $\mathtt{true}_1$, $\mathtt{false}_1$ and $\mathtt{true}_2$, $\mathtt{false}_2$ respectively, then it is unsound to apply the conflict rule on the (heterogeneous) equation $\mathtt{true}_1 = \mathtt{false}_2$. Doing so would allow us to prove that $\mathtt{Bool}_1 \not\simeq \mathtt{Bool}_2$, contradicting univalence. The unification algorithm used by Agda 2.4 (and older) contains a number of ad hoc restrictions to the unification algorithm to avoid bad unification steps like this, but these restrictions had no theoretical ground and didn't handle every case correctly.

Example 3.3. The problem isn't limited to theories that don't support UIP, either. Let A be an arbitrary type and $\texttt{Singleton} : A \to \texttt{Set}$ be an indexed datatype with one constructor $\texttt{sing} : (x : A) \to \texttt{Singleton}\ x$ and consider the unification problem $(\texttt{Singleton}\ s, \texttt{sing}\ s) = (\texttt{Singleton}\ t, \texttt{sing}\ t)$. If we allow the injectivity rule to simplify $\texttt{sing}\ s = \texttt{sing}\ t$ to $s = t$ then we could prove injectivity of the type constructor $\texttt{Singleton}$. In particular, if we take $A = \texttt{Set} \to \texttt{Set}$ then this injectivity allows us to refute the law of the excluded middle (Theorem 1.22). In general, injectivity of type constructors is an undesirable property because it is not only incompatible with the law of the excluded middle but also with univalence (Theorem 1.21) and with an impredicative universe of propositions (Miquel, 2010).

To avoid these problems, we give a fully typed account of unification of dependently typed data where we treat unification problems and unification rules as internal to the type theory. First, we represent the input of the unification problem by a *telescopic equality* where each type in this telescope corresponds to one equation of the unification problem (Section 3.1.1). Next, we represent the output of the unification algorithm by an equivalence between the original telescope of equations and a trivial one (Section 3.1.2). This equivalence contains not only the substitution computed by the unification algorithm, but also *evidence* that the output is correct. Finally, the unification algorithm itself then works by successively applying unification rules, which are represented by equivalences between two telescopes of equations (Section 3.1.3).

3.1.1 Unification problems as telescopes

To represent unification problems internally, we need to express equality between two terms as a type. For this purpose, we use the propositional equality type $x \equiv_A y$. For example, the unification problem $\texttt{suc}\ m = \texttt{suc}\ n$ is represented by the type $\texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n$.

The identity type $x \equiv_A y$ only allows equations between elements of the same type, so we still need a way to represent heterogeneous equations. For this purpose, McBride (2000) introduced a heterogeneous equality type $x\ {}_A{\cong}_B\ y$ where $x : A$ and $y : B$ can be of different types, but $x\ {}_A{\cong}_B\ y$ can only be proven if the types A and B are actually the same. Using this type, a unification problem can be represented by the (non-dependent) product of the individual equalities. By maintaining the invariant that the leftmost equation is always homogeneous, the equations can be solved step by step, from left to right. However, using this heterogeneous equality type causes a number of problems:

- Turning a heterogeneous equation between elements of the same type into a homogeneous one requires the K rule. So in a theory without the general K rule (such as HoTT), heterogeneous equalities are worthless.
- Using heterogeneous equality causes information about dependencies between the equations to be lost. For example, if we have two equations $\texttt{Bool} \ {}_{\texttt{Set}}{\cong}_{\texttt{Set}} \ \texttt{Bool}$ and $\texttt{true} \ {}_{\texttt{Bool}}{\cong}_{\texttt{Bool}} \ \texttt{false}$, there is no way to see whether the type of the second equation depends on the first. Example 3.1 shows that both cases are possible, and that it is essential to know the difference!
- Finally, it is unsound to postpone an equation and continue with the next one when working with heterogeneous equality, since this allows us to prove things such as injectivity of certain type constructors (Example 3.3).

To avoid these problems and keep track of the dependencies between equations, we use the concept from HoTT of an equality "laying over" another one. There are multiple equivalent ways to define this type; for the sake of simplicity we use the following definition in terms of the regular homogeneous equality by substituting on the left:

Definition 3.4 (Dependent identity type)**.** Let $e : s \equiv_A t$ and $P : (x : A) \to \texttt{Set}$. We define the type $u \equiv^e_P v$ of equality proofs between $u : P\ s$ and $v : P\ t$ *laying over* e by

$$u \equiv^e_P v = (\texttt{subst}\ P\ e\ u) \equiv_{(P\ t)} v \tag{3.1}$$

There are other possible definitions of $u \equiv^e_P v$, for example it can be defined as a new datatype with one constructor $\texttt{refl'} : u \equiv^{\texttt{refl}}_P u$, or it can be defined as a function by pattern matching on the proof e. In practice, the exact definition doesn't have much impact, what's important is that $(u \equiv^{\texttt{refl}}_P v) = (u \equiv_{P\ s} v)$ whenever $s = t$. We prefer the definition here to the more symmetric alternatives because it doesn't require large eliminations or auxiliary datatypes.

We often write $u \equiv_{P\ e} v$ instead of $u \equiv^e_P v$. For example, if $e : m \equiv_{\mathbb{N}} n$ and $u : \texttt{Vec}\ A\ m$ and $v : \texttt{Vec}\ A\ n$ are two vectors, then we may form the type $u \equiv_{\texttt{Vec}\ A\ e} v$. This notation is inspired by cubical type theory (Cohen et al., 2016), where a function $f : A \to B$ is automatically lifted to a function $x \equiv_A y \to f\ x \equiv_B f\ y$. In our setting it is merely a convenient abuse of notation.

Using this notion of an equality proof laying over another, we can define a version of `cong` that works for dependent functions:

$$\begin{aligned}&\texttt{dcong} : (f : (x : A) \to B\ x) \to \{x\ y : A\} \to (e : x \equiv_A y) \to f\ x \equiv_{B\ e} f\ y\\ &\texttt{dcong}\ f\ \texttt{refl} = \texttt{refl}\end{aligned} \tag{3.2}$$

In general, an equality may depend on more than one equation variable. To keep track of the types P and the equation variables e, we give a type to the list of equations in the form of a *telescope*. Telescopic equality is defined as follows:

Definition 3.5 (Telescopic equality)**.** Let Δ be a telescope and $\bar{s}, \bar{t} : \Delta$. We define telescopic equality $(\bar{e} : \bar{s} \equiv_\Delta \bar{t})$ inductively on the length of the telescope by $() \equiv_{()} () = ()$ and

$$(e; \bar{e} : (s; \bar{s}) \equiv_{(x:A)\Delta} (t; \bar{t})) = (e : s \equiv_A t)(\bar{e} : \bar{s} \equiv_{\Delta[x \mapsto e]} \bar{t}) \tag{3.3}$$

For each $\bar{t} : \Delta$, we define $\overline{\texttt{refl}} : \bar{t} \equiv_\Delta \bar{t}$ as $(\texttt{refl}; \ldots; \texttt{refl})$.

For example, $(e_1; e_2 : (m; u) \equiv_{(x:\mathbb{N})(y:\texttt{Vec}\ A\ x)} (n; v))$ stands for the telescope $(e_1 : m \equiv_{\mathbb{N}} n)(e_2 : u \equiv_{\texttt{Vec}\ A\ e_1} v)$.

Definition 3.6 (Unification problem)**.** A *unification problem* is a telescope of the form $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. The variables in Γ are called the *flexible variables*.

Example 3.7. A unification problem between two vectors can be represented by the telescope

$$\begin{aligned}&(m\ n : \mathbb{N})(x\ y : A)(xs : \texttt{Vec}\ A\ m)(ys : \texttt{Vec}\ A\ n)\\ &(e_1 : \texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n)(e_2 : \texttt{cons}\ x\ xs \equiv_{\texttt{Vec}\ A\ e}\ \texttt{cons}\ y\ ys)\end{aligned} \tag{3.4}$$

Lemma 3.8 ($\overline{\mathrm{J}}$)**.** *We have the telescopic equality eliminator*

$$\overline{\mathrm{J}} : (P : (\bar{s} : \Delta) \to \bar{r} \equiv \bar{s} \to \texttt{Set}_i) \to P\ \bar{r}\ \overline{\texttt{refl}} \to (\bar{s} : \Delta) \to (\bar{e} : \bar{r} \equiv \bar{s}) \to P\ \bar{s}\ \bar{e} \tag{3.5}$$

Construction. We define $\overline{\mathrm{J}}$ by eliminating the equations $\bar{e}$ from left to right using J:

$$\begin{aligned}\overline{\mathrm{J}}\ P\ p\ ()\ \ \ ()\ \ \ &= p\\ \overline{\mathrm{J}}\ P\ p\ (s; \bar{s})\ (e; \bar{e})\ &= \mathrm{J}\ (\lambda s; e.\ (\bar{s} : \Delta)(\bar{e} : \bar{r} \equiv \bar{s}) \to P\ (s; \bar{s})\ (e; \bar{e}))\\ &\quad\ (\lambda \bar{s}; \bar{e}.\ \overline{\mathrm{J}}\ (\lambda \bar{s}; \bar{e}.\ P\ (r; \bar{s})\ (\texttt{refl}; \bar{e}))\ p\ \bar{e})\\ &\quad\ e\ \bar{s}\ \bar{e}\end{aligned} \tag{3.6}$$

Each elimination of an equation $e_i : r_i \equiv s_i$ fills in $\texttt{refl}$ for all occurrences of e_i, allowing the next equations to reduce and in particular ensuring that the following equation is of the correct form. □

Using $\overline{\mathrm{J}}$, we also define telescopic versions of `subst`, `cong` and `dcong`:

$$\begin{aligned}\overline{\texttt{subst}}\ &:\ (P : \Delta \to \texttt{Set}_\ell) \to \{\bar{u}\ \bar{v} : \Delta\} \to \bar{u} \equiv_\Delta \bar{v} \to P\ \bar{u} \to P\ \bar{v}\\ \overline{\texttt{cong}}\ &:\ (f : \Delta \to T) \to \{\bar{u}\ \bar{v} : \Delta\} \to \bar{u} \equiv_\Delta \bar{v} \to f\ \bar{u} \equiv_T f\ \bar{v}\\ \overline{\texttt{dcong}}\ &:\ (f : (\bar{x} : \Delta) \to T\ \bar{x}) \to \{\bar{u}\ \bar{v} : \Delta\} \to (\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to f\ \bar{u} \equiv_{T\ \bar{e}} f\ \bar{v}\end{aligned} \tag{3.7}$$

3.1.2 Unifiers as equivalences

Traditionally, a unifier for a unification problem $\bar{u} = \bar{v}$ is defined as a substitution σ such that $\bar{u}\sigma$ and $\bar{v}\sigma$ are equal. So how do we translate this definition to type theory? Consider a unification problem of the form $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. We could represent a unifier as a telescope mapping $\sigma : \Gamma' \to \Gamma$ satisfying $\bar{u}[\Gamma \mapsto \sigma\ \Gamma] = \bar{v}[\Gamma \mapsto \sigma\ \Gamma]$, but then the correctness property is still external to the theory. Instead, we use the power of dependent types to express the fact that the equations are satisfied internally:

Definition 3.9 (Unifier)**.** Let Γ and Δ be telescopes and $\bar{u}$ and $\bar{v}$ be lists of terms such that $\Gamma \vdash \bar{u}, \bar{v} : \Delta$. We define a *unifier* of $\bar{u}$ and $\bar{v}$ as a telescope mapping $\sigma : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ for some Γ'.

A unifier σ returns not only values for Γ but also *evidence* that the equations are indeed satisfied by these values.

Usually, the goal of a unification algorithm is not just to output any unifier but a *most general* one, i.e. a unifier $\sigma : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_A \bar{v})$ such that any other unifier $\sigma' : \Gamma'' \to \Gamma(\bar{e} : \bar{u} \equiv_A \bar{v})$ can be written as $\sigma \circ \nu$ for some $\nu : \Gamma'' \to \Gamma'$.

Again, we should think how to represent this concept internally. One way to do this is to translate the definition of most general unifier directly to a type. However, to do this we need to quantify over all possible telescopes Γ'' and unifiers σ', making the definition more unwieldy than necessary. Can we find a better definition?

Definition 3.10 (Pointwise equality)**.** Let $f, g : (x : A) \to B\ x$ be two functions. The pointwise equality type $f \doteq g$ is defined as $(x : A) \to f\ x \equiv_{B\ x} g\ x$. Similarly, if $\sigma, \tau : \Delta \to \Gamma$ are two telescope mappings then $\sigma \doteq \tau$ is defined as $(\bar{x} : \Delta) \to \sigma\ \bar{x} \equiv_\Gamma \tau\ \bar{x}$.

Lemma 3.11. *Let Γ and Γ' be telescopes and $\sigma : \Gamma' \to \Gamma$. The following two statements are equivalent:*

- *For any telescope Γ'' and $\sigma' : \Gamma'' \to \Gamma$, there exists a $\nu : \Gamma'' \to \Gamma'$ such that $\sigma' \doteq \sigma \circ \nu$.*
- *There exists a $\tau_1 : \Gamma \to \Gamma'$ such that $\sigma \circ \tau_1 \doteq \mathtt{id}$.*

Proof. First suppose that we have a telescope mapping $\tau_1 : \Gamma \to \Gamma'$ such that $\sigma \circ \tau_1 \doteq \mathtt{id}$ is the identity function on Γ. This allows us to define $\nu = \tau_1 \circ \sigma'$, which gives us $\sigma \circ \nu \doteq \sigma \circ \tau_1 \circ \sigma' \doteq \sigma'$, as we wanted.

For the other direction, we take $\Gamma'' = \Gamma$ and $\sigma' = \mathtt{id}$. Then by assumption we have a $\tau_1 : \Gamma \to \Gamma'$ such that $\mathtt{id} \doteq \sigma \circ \tau_1$. □

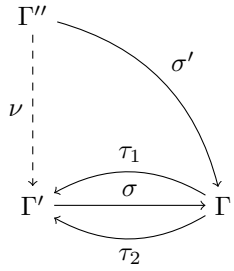


Figure 3.1: Lemma 3.11 allows us to construct a right inverse τ_1 to σ from the existence of the telescope mapping ν, while Lemma 3.12 gives us a left inverse τ_2 from its uniqueness.

It is often useful to require that the function ν is unique, for otherwise Γ' may contain ghost variables that are not actually used by σ. For example, for a unification problem with $\Gamma = (b : \texttt{Bool})$ and a single equation $b \equiv_{\texttt{Bool}} \texttt{true}$, we have the most general unifier $\sigma : () \to (b : \texttt{Bool})(e : b \equiv_{\texttt{Bool}} \texttt{true})$. However, if ν is not required to be unique, then there may be other most general unifiers with a non-equivalent choice of Γ'. For example, we could also have taken $\sigma' : (b' : \texttt{Bool}) \to (b : \texttt{Bool})(e : b \equiv_{\texttt{Bool}} \texttt{true})$ that ignores its argument b'.

Lemma 3.12. *The telescope mapping ν constructed in Lemma 3.11 is unique (up to pointwise equality) if and only if there exists a $\tau_2 : \Gamma \to \Gamma'$ such that $\tau_2 \circ \sigma \doteq \texttt{id}$.*

Proof. Suppose that we have a τ_2 such that $\tau_2 \circ \sigma \doteq \texttt{id}$. If ν and ν' are two telescope mappings such that $\sigma \circ \nu \doteq \sigma' \doteq \sigma \circ \nu'$ then we have $\nu \doteq \tau_2 \circ \sigma \circ \nu \doteq \tau_2 \circ \sigma \circ \nu' \doteq \nu'$, so ν is unique.

For the other direction, we assume that ν is unique. Let $\Gamma'' = \Gamma'$ and $\sigma' = \sigma$ and $\nu = \tau_1 \circ \sigma$ and $\nu' = \texttt{id}$. This gives us that $\sigma \circ \nu \doteq \sigma \doteq \sigma \circ \nu'$, so by uniqueness we have $\tau_1 \circ f \doteq \texttt{id}$. Hence taking $\tau_2 = \tau_1$ gives us the desired telescope mapping τ_2. □

The proofs of Lemma 3.11 and Lemma 3.12 are illustrated in Figure 3.1.

If we replace the telescope Γ by a unification problem $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$, then Lemma 3.11 and Lemma 3.12 together give us that $\sigma : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to \Gamma'$ is a most general unifier if and only if it is an *equivalence* between Γ' and $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$.

This brings us to the following definition of a most general unifier:

Definition 3.13 (Most general unifier)**.** Let Γ and Δ be telescopes and $\bar{u}$ and $\bar{v}$ be lists of terms such that $\Gamma \vdash \bar{u}, \bar{v} : \Delta$. Then a *most general unifier* of $\bar{u}$ and $\bar{v}$ is an equivalence $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ for some telescope Γ'.

The unifier $\sigma : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ corresponds to the inverse function f^{-1}. Intuitively, f allows us to recover the values of the variables in Γ' for any values of Γ that satisfy $\bar{u} \equiv_\Delta \bar{v}$.

The definition of a most general unifier doesn't prevent us from choosing $\Gamma' = \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ and $f = \mathtt{id}$. In fact, this is a valid (if trivial) most general unifier from a logical point of view. However, it doesn't satisfy the requirements for a strong unifier (Definition 3.45).

In case unification succeeds negatively, we need evidence that the equations are indeed impossible. For this purpose, we use the empty type $\bot$:

Definition 3.14 (Disunifier)**.** Let Γ and Δ be telescopes and $\bar{u}$ and $\bar{v}$ be lists of terms such that $\Gamma \vdash \bar{u}, \bar{v} : \Delta$. A *disunifier* of $\bar{u}$ and $\bar{v}$ is an equivalence $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$.

Any function $f : A \to \bot$ is automatically an equivalence $A \simeq \bot$, as the other components of the equivalence can be constructed by using the eliminator $\mathtt{elim}_\bot : (A : \mathtt{Set}_\ell) \to \bot \to A$. So the only interesting part of a disunifier is the function $f : A \to \bot$.

3.1.3 The unification algorithm

Now that we know how to represent the input and the output of the unification algorithm, we can start thinking about the unification algorithm itself. Since the end result of the unification process (the most general unifier) is an equivalence, it is natural to represent unification rules as equivalences as well. These unification rules can then be chained together by transitivity of $\simeq$ to produce the most general unifier f.

Definition 3.15 (Positive unification rule)**.** A *positive unification rule* is an equivalence of the form $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$.

For example, the unification rule for injectivity of the $\mathtt{suc}$ constructor for $\mathbb{N}$ is $\mathtt{injectivity}_{\mathtt{suc}} : (e : \mathtt{suc}\ m \equiv \mathtt{suc}\ n) \simeq (e' : m \equiv n)$.

In addition to unification rules of this form, that transform one set of equations into another, there are also unification rules that refute absurd equations like $\mathtt{true} \equiv_{\mathtt{Bool}} \mathtt{false}$.

Definition 3.16 (Negative unification rule)**.** A *negative unification rule* is an equivalence of the form $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$.

For example, the unification rule for conflict between `true` and `false` is $\texttt{conflict}_{\texttt{true},\texttt{false}} : (\texttt{true} \equiv_{\texttt{Bool}} \texttt{false}) \simeq \bot$.

The unification algorithm tries to construct an equivalence $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ by successively applying the unification rules to the unification problem, simplifying one or more equations in each step. This process continues until one of three possible situations occurs:

- If there are no more equations left, the algorithm succeeds positively. In this case, it returns an equivalence between the original problem $\Gamma(\bar{u} \equiv_\Delta \bar{v})$ and the reduced telescope Γ'.
- If a contradictory equation is encountered, the algorithm succeeds negatively. In this case, it returns an equivalence between the original problem $\Gamma(\bar{u} \equiv_\Delta \bar{v})$ and the empty type $\bot$.
- If there are no more applicable rules, the algorithm results in a failure.

We don't yet give an explicit strategy on which rule to apply in a specific situation. This leaves more freedom to the implementation to choose which rule to try first. When we discuss our implementation in Section 3.5, we give one concrete strategy.

Example 3.17. Consider the unification problem consisting of flexible variables $k\ l : \mathbb{N}$ and a single equation between $\texttt{suc}\ k$ and $\texttt{suc}\ l$. First, we simplify the equation by applying the equivalence $\texttt{injectivity}_{\texttt{suc}} : (e : \texttt{suc}\ k \equiv_\mathbb{N} \texttt{suc}\ l) \simeq (e : k \equiv_\mathbb{N} l)$ (Lemma 3.23). Applying this rule leaves the two variables k and l unchanged. Next, we apply the `solution` rule (Lemma 3.21), which tells us that $(l : \mathbb{N})(e : k \equiv_\mathbb{N} l) \simeq ()$. This leaves only the single variable $k : \mathbb{N}$. Since there are no more equations left in the telescope, unification ends in a positive success.

$$\begin{aligned}(k\ l : \mathbb{N})(\underline{e} : \texttt{suc}\ k \equiv_\mathbb{N} \texttt{suc}\ l)\\ \simeq (k\ \underline{l} : \mathbb{N})(\underline{e} : k \equiv_\mathbb{N} l)\\ \simeq (k : \mathbb{N})\end{aligned} \tag{3.8}$$

To get the substitution from $(k : \mathbb{N})$ to $(k\ l : \mathbb{N})(e : \texttt{suc}\ k \equiv_\mathbb{N} \texttt{suc}\ l)$ computed by the unification process, we only need to compose the functions embedded in the equivalences from the bottom to the top. The solution rule assigns $l \mapsto k$ and $e \mapsto \texttt{refl}$, and the injectivity rule maps $e : k \equiv_\mathbb{N} l$ to $\texttt{cong}\ \texttt{suc}\ e : \texttt{suc}\ k \equiv_\mathbb{N} \texttt{suc}\ l$, so the complete substitution is $k \mapsto (k; k; \texttt{refl})$ (since $\texttt{cong}\ \texttt{suc}\ \texttt{refl} = \texttt{refl}$).

Before we continue with the general form of the unification rules in the next section, we first give three easy but useful manipulations on equivalences (and hence on unification rules) that allow us to postpone and reorder equations. These principles are used when we want to apply a unification rule, but the problem contains some additional variables or equations that aren't mentioned in the rule.

Lemma 3.18 (f_Γ)**.** *If we have an equivalence* $f : \Delta \simeq \Delta'$ *where* Δ *and* Δ' *possibly contain free variables from a telescope* Γ*, then we also have an equivalence* $f_\Gamma : \Gamma\Delta \simeq \Gamma\Delta'$*.*

Lemma 3.19 (f^Δ)**.** *If we have an equivalence* $f : \Gamma \simeq \Gamma'$ *and a telescope* Δ *possibly containing free variables from* Γ*, then we also have an equivalence* $f^\Delta : \Gamma\Delta \simeq \Gamma'\Delta'$ *where* $\Delta' = \Delta[\Gamma \mapsto f^{-1}\ \Gamma']$*.*

Lemma 3.20. *If we have a telescope* Γ*, and* Γ' *is a reordering of the variable bindings in* Γ *that preserves the order of dependencies, then we have an equivalence* $f : \Gamma \simeq \Gamma'$*.*

Construction. The construction of the equivalence is in all three cases straightforward, relying on J to prove that the functions are mutual inverses. □

In what follows, we often make use of these manipulations implicitly. When we do so, we underline the variables that are actually mentioned by the unification rule. The remaining variables are handled by the three above lemma's. For example, in Example 3.17, Lemma 3.18 is applied to lift the equivalence $\texttt{injectivity}_{\texttt{suc}} : (e : \texttt{suc}\ k \equiv_{\mathbb{N}} \texttt{suc}\ l) \simeq (e : k \equiv_{\mathbb{N}} l)$ into an equivalence $(k\ l : \mathbb{N})(\underline{e} : \texttt{suc}\ k \equiv_{\mathbb{N}} \texttt{suc}\ l)$, and to lift $\texttt{solution} : (l : \mathbb{N})(e : k \equiv_{\mathbb{N}} l) \simeq ()$ to $(k\ \underline{l} : \mathbb{N})(\underline{e} : k \equiv_{\mathbb{N}} l) \simeq (k : \mathbb{N})$.

3.2 Unification rules

In this section, we state the basic unification rules from McBride (1998b) in our framework. We first handle the unification rules for simple datatypes (Section 3.2.1) before moving on to the more challenging rules for indexed datatypes (Section 3.2.2). We also show how to extend the unification algorithm with rules for η-equality for record types (Section 3.2.3). Finally, we study the computational behaviour of these unification rules as type-theoretic terms (Section 3.3).

3.2.1 The basic unification rules

The first two rules are generic in the sense that they work for any type A.

Lemma 3.21 (Solution)**.** *For any type A and term $t : A$, we have an equivalence*

$$\texttt{solution} : (x : A)(e : x \equiv_A t) \simeq () \tag{3.9}$$

satisfying $\texttt{solution}^{-1}\ () = (t; \texttt{refl})$.

In this rule, the variable x should not occur freely in t.

Construction of `solution`*.* The construction of the functions $\texttt{solution} : (x : A)(x \equiv_A t) \to ()$ and $\texttt{isRinv solution} : () \to () \equiv_{()} ()$ is trivial since they both target an empty telescope. The function $\texttt{solution}^{-1} : () \to (x : A)(e : x \equiv_A t)$ is defined by $\texttt{solution}^{-1}\ () = (t; \texttt{refl})$. Finally, $\texttt{isLinv solution} : (x : A)(e : x \equiv_A t) \to (t; \texttt{refl}) \equiv_{(x:A)(e:x\equiv_A t)} (x; e)$ is a direct application of the J rule. □

Lemma 3.22 (Deletion)**.** *For any type A that satisfies* K *and any term $t : A$, we have an equivalence*

$$\texttt{deletion} : (e : t \equiv_A t) \simeq () \tag{3.10}$$

Construction of `deletion`*.* The construction is similar to the construction of `solution`, except that K is used instead of J. □

The **injectivity**, **conflict**, and **cycle** rules are specific to an inductive datatype `D`. We present them here for a simple (non-indexed) datatype and for an indexed datatype in the next section.

Lemma 3.23 (Injectivity, simply-typed version)**.** *Let* $\texttt{c} : \Delta_\texttt{c} \to \texttt{D}$ *be a constructor of the datatype* $\texttt{D} : \texttt{Set}_\ell$ *and let* $\bar{s}, \bar{t} : \Delta_\texttt{c}$*. We have an equivalence*

$$\texttt{injectivity}_\texttt{c} : (\texttt{c}\ \bar{s} \equiv_\texttt{D} \texttt{c}\ \bar{t}) \simeq (\bar{s} \equiv_{\Delta_\texttt{c}} \bar{t}) \tag{3.11}$$

such that $\texttt{injectivity}_\texttt{c}^{-1}\ \bar{e} = \overline{\texttt{cong}}\ \texttt{c}\ \bar{e}$.

Lemma 3.24 (Conflict, simply-typed version)**.** *Let* $\texttt{c}_1 : \Delta_1 \to \texttt{D}$ *and* $\texttt{c}_2 : \Delta_2 \to \texttt{D}$ *be two distinct constructors of the datatype* $\texttt{D} : \texttt{Set}_\ell$ *and* $\bar{s} : \Delta_1$ *and* $\bar{t} : \Delta_2$*. We have an equivalence*

$$\texttt{conflict}_{\texttt{c}_1,\texttt{c}_2} : (\texttt{c}_1\ \bar{s} \equiv_\texttt{D} \texttt{c}_2\ \bar{t}) \simeq \bot \tag{3.12}$$

Lemma 3.25 (Cycle, simply-typed version)**.** *Let* $\mathtt{D} : \mathtt{Set}_\ell$ *be a datatype and let* $s, t : \mathtt{D}$ *be such that* $s \prec t$*. We have an equivalence*

$$\mathtt{cycle}_{s,t} : (s \equiv_{\mathtt{D}} t) \simeq \bot \tag{3.13}$$

Once again, the type of the equation should be exactly $\mathtt{D}$.

Construction of $\mathtt{injectivity}_{\mathtt{c}}$*,* $\mathtt{conflict}_{\mathtt{c}_1,\mathtt{c}_2}$ *and* $\mathtt{cycle}_{x,t}$*.* See Lemma 4.16 for the construction of $\mathtt{injectivity}_{\mathtt{c}}$ and $\mathtt{conflict}_{\mathtt{c}_1,\mathtt{c}_2}$ and Lemma 4.20 for the construction of $\mathtt{cycle}_{x,t}$. □

Example 3.26. Consider the sum type $A \uplus B$ (where $A, B : \mathtt{Set}$ are arbitrary types) with two constructors $\mathtt{left} : A \to A \uplus B$ and $\mathtt{right} : B \to A \uplus B$. An expression of the form $\mathtt{left}\ x$ is never equal to $\mathtt{right}\ y$, so any equality between those two terms is equivalent to $\bot$:

$$(x : A)(y : B)(\underline{e} : \mathtt{left}\ x \equiv_{A \uplus B} \mathtt{right}\ y) \simeq \bot \tag{3.14}$$

This is exactly the conflict rule between $\mathtt{left}$ and $\mathtt{right}$.

The type of the equation on the left in Lemma 3.23 and Lemma 3.24 should be exactly $\mathtt{D}$, in particular the constructor $\mathtt{c}$ must be fully applied.

Counterexample 3.27. An equation between the constructors $\mathtt{left}$ and $\mathtt{right}$ is not always absurd when they are not fully applied. Let $A = B = \bot$, then $(e : \mathtt{left} \equiv_{\bot \to \bot \uplus \bot} \mathtt{right})$ is not equivalent to $\bot$. This is because when viewed as functions of type $\bot \to \bot \uplus \bot$, the constructors $\mathtt{inj}_1$ and $\mathtt{inj}_2$ coincide on all possible inputs (i.e. none). The principle of functional extensionality then tells us that these two functions are equal. So if we would consider this equation to be absurd, we would prohibit ourselves from having functional extensionality in our language, nevertheless a desirable property to have! Wrongly applying the conflict rule in this way led to the problem described by issue #1497 on the Agda bug tracker (Dijkstra, 2015).

3.2.2 Rules for indexed datatypes

The injectivity, conflict, and cycle rules defined in the previous section all work on regular datatypes, but unification only becomes really interesting once we consider indexed families of datatypes. Where the unification rules that we have seen so far only have a single equation on the left side, the rules for indexed datatypes have a telescope of equations: one equation for each index, and one final equation for the datatype itself. If $\mathtt{D}$ is a datatype with indices Ξ, then the telescope for this sequence of equations is $\overline{\mathtt{D}} = (\bar{u} : \Xi)(x : \mathtt{D}\ \bar{u})$.

Lemma 3.28 (Injectivity, general version)**.** *Let* $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$ *be a constructor of the datatype* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$*. Then we have an equivalence*

$$\mathtt{injectivity_c} : ((\bar{u}[\Delta \mapsto \bar{s}]; \mathtt{c}\ \bar{s}) \equiv_{\overline{\mathtt{D}}} (\bar{u}[\Delta \mapsto \bar{t}]; \mathtt{c}\ \bar{t})) \simeq (\bar{s} \equiv_\Delta \bar{t}) \tag{3.15}$$

where $\bar{s}, \bar{t} : \Delta$*. Moreover,* $\mathtt{injectivity_c}$ *satisfies* $\mathtt{injectivity_c}^{-1}\ \bar{e} = \overline{\mathtt{dcong}}\ (\lambda \bar{x}.\ (\bar{u}; \mathtt{c}\ \bar{x}))\ \bar{e}$*.*

Lemma 3.29 (Conflict, general version)**.** *Let* $\mathtt{c}_1 : \Delta_1 \to \mathtt{D}\ \bar{u}_1$ *and* $\mathtt{c}_2 : \Delta_2 \to \mathtt{D}\ \bar{u}_2$ *be two distinct constructors of the datatype* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$*. Then we have an equivalence*

$$\mathtt{conflict_{c_1,c_2}} : ((\bar{u}_1[\Delta_1 \mapsto \bar{s}]; \mathtt{c}_1\ \bar{s}) \equiv_{\overline{\mathtt{D}}} (\bar{u}_2[\Delta_2 \mapsto \bar{t}]; \mathtt{c}_2\ \bar{t})) \simeq \bot \tag{3.16}$$

where $\bar{s} : \Delta_1$ *and* $\bar{t} : \Delta_2$*.*

Lemma 3.30 (Cycle, general version)**.** *Let* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ *be a datatype and let* $(\bar{u}; s), (\bar{v}; t) : \overline{\mathtt{D}}$ *be such that* $s \prec t$*. Then we have an equivalence*

$$\mathtt{cycle}_{x,t} : ((\bar{u}; s) \equiv_{\overline{\mathtt{D}}} (\bar{v}; t)) \simeq \bot \tag{3.17}$$

Construction of $\mathtt{injectivity_c}$*,* $\mathtt{conflict_{c_1,c_2}}$ *and* $\mathtt{cycle}_{x,t}$*.* See Lemma 4.16 for the construction of $\mathtt{injectivity_c}$ and $\mathtt{conflict_{c_1,c_2}}$ and Lemma 4.20 for the construction of $\mathtt{cycle}_{x,t}$. □

Example 3.31. Consider the indexed datatype $\mathtt{Vec}\ A : \mathbb{N} \to \mathtt{Set}$ with the two constructors $\mathtt{nil} : \mathtt{Vec}\ A\ \mathtt{zero}$ and $\mathtt{cons} : (n : \mathbb{N}) \to A \to \mathtt{Vec}\ A\ n \to \mathtt{Vec}\ A\ (\mathtt{suc}\ n)$. The injectivity rule for cons gives us the following equivalence:

$$\begin{aligned} &((\mathtt{suc}\ m; \mathtt{cons}\ m\ x\ xs) \equiv_{\overline{\mathtt{Vec}\ A}} (\mathtt{suc}\ n; \mathtt{cons}\ n\ y\ ys)) \\ &\quad\simeq ((m; x; xs) \equiv_{(n:\mathbb{N})(x:A)(xs:\mathtt{Vec}\ A\ n)} (n; y; ys)) \end{aligned} \tag{3.18}$$

This rule not only simplifies the equation between the two cons constructors, but simultaneously simplifies the equation between the indices $\mathtt{suc}\ m$ and $\mathtt{suc}\ n$. Now let's see how this rule works in action:

$$\begin{aligned} &(m\ n : \mathbb{N})(x\ y : A)(xs : \mathtt{Vec}\ A\ m)(ys : \mathtt{Vec}\ A\ n) \\ &(\underline{e_1} : \mathtt{suc}\ m \equiv_{\mathbb{N}} \mathtt{suc}\ n)(\underline{e_2} : \mathtt{cons}\ m\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys) \\ &\quad\simeq \begin{array}{l}(\underline{m}\ n : \mathbb{N})(x\ y : A)(xs : \mathtt{Vec}\ A\ m)(ys : \mathtt{Vec}\ A\ n) \\ (\underline{e_1} : m \equiv_{\mathbb{N}} n)(\underline{e_2} : x \equiv_A y)(\underline{e_3} : xs \equiv_{\mathtt{Vec}\ A\ e_1} ys)\end{array} \\ &\quad\simeq (n : \mathbb{N})(x : A)(xs : \mathtt{Vec}\ A\ n) \end{aligned} \tag{3.19}$$

The first step is an application of the injectivity rule, while the next step consists of three applications of the solution rule.

To apply injectivity of cons, the type of the equation has to be of the form Vec A e where e refers to a previous equation. This implies that this rule cannot be applied directly to an equation of the form cons n x $xs \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)}$ cons n y ys where xs : Vec A n and ys : Vec A n have the same length 'on the nose'. We discuss how to solve this deficiency in Section 3.4.

Example 3.32. In the previous example, it was not really necessary to simplify the equation between the indices together with the equation between the constructors, as we could also have applied $\texttt{injectivity}_{\texttt{suc}}$ to the equation suc $m \equiv_{\mathbb{N}}$ suc n. However, sometimes this simplification gives a real increase to the power of unification. For example, let $f : A \to B$ be a (possibly very complex) function, then in general there is no way to solve an equation of the form $f\ x \equiv_B f\ y$. Now consider the following datatype

$$\begin{aligned}&\textbf{data}\ \texttt{Im}\ f : B \to \texttt{Set}\ \textbf{where}\\ &\quad \texttt{image} : (x : A) \to \texttt{Im}\ f\ (f\ x)\end{aligned} \tag{3.20}$$

The injectivity rule for image simultaneously solves the equations $e_1 : f\ x \equiv_B f\ y$ and e_2 : image $x \equiv_{\texttt{Im}\ f\ e_1}$ image y:

$$\begin{aligned}&(x\ y : A)(\underline{e_1} : f\ x \equiv_B f\ y)(\underline{e_2} : \texttt{image}\ x \equiv_{\texttt{Im}\ f\ e_1} \texttt{image}\ y)\\ &\quad \simeq (x\ \underline{y} : A)(\underline{e} : x \equiv_A y)\\ &\quad \simeq (x : A)\end{aligned} \tag{3.21}$$

Remember that the variables that are underlined are handled by the unification rules themselves ($\texttt{injectivity}_{\texttt{image}}$ and solution in this case), while the remaining variables are handled by Lemma 3.18, Lemma 3.19, and Lemma 3.20. Having an injectivity rule that works in this way is useful when giving semantics to an embedded language (Danielsson, 2015).

Contrast this with the unification problem

$$(x\ y : A)(e_1 : \texttt{Im}\ f\ (f\ x) \equiv_{\texttt{Set}} \texttt{Im}\ f\ (f\ y))(e_2 : \texttt{image}\ x \equiv_{e_1} \texttt{image}\ y) \tag{3.22}$$

Here, it is not allowed to use injectivity on the second equation since its type is not a datatype but a variable. Like in Example 3.1, there is no way to distinguish between these two cases unless we keep track of the dependency of the type of e_2 on the equation e_1. Wrongly applying injectivity in situations like this led to the problems described by Abel (2015a,c) on the Agda bug tracker.

Example 3.33. Let D : Bool → Set be an indexed datatype with two constructors tt : D true and ff : D false. Then the conflict rule between tt and ff gives us the following equivalence:

$$(e_1 : \texttt{true} \equiv_{\texttt{Bool}} \texttt{false})(e_2 : \texttt{tt} \equiv_{\texttt{D}\ e_1} \texttt{ff}) \simeq \bot \tag{3.23}$$

In contrast with the above example, the conflict rule cannot be applied if the first equation is between the *types* `D true` and `D false`:

$$(e_1 : \texttt{D true} \equiv_{\texttt{Set}} \texttt{D false})(e_2 : \texttt{tt} \equiv_{e_1} \texttt{ff}) \not\simeq \bot \tag{3.24}$$

Allowing the conflict rule to apply in this case would mean that we can distinguish between `D true` and `D false`, which means that the type constructor `D` is injective. In particular, this would be incompatible with univalence: there is an equivalence between `D true` and `D false` under which `tt` is identified with `ff`, so univalence allows us to prove that `D true` $\equiv_{\texttt{Set}}$ `D false`. Note again that we need information about how the type of e_2 depends on e_1 to distinguish between these two cases. Wrongly applying conflict in situations like this led to the problems described by Danielsson (2010) and Vezzosi (2015) on the Agda bug tracker.

Example 3.34. This example is based on issue #1071 on the Agda bug tracker (Danielsson, 2014). Let $A : \texttt{Set}$ and $F : \texttt{Set} \to \texttt{Set}$ and $\texttt{P} : \texttt{Set} \to \texttt{Set}_1$ be a datatype with one constructor $\texttt{c} : (A : \texttt{Set}) \to \texttt{P}\ (F\ A)$. Then we have:

$$\begin{aligned}
&(f : F\ A)(R : \texttt{Set})(f' : F\ R)\\
&(e_1 : F\ A \equiv_{\texttt{Set}} F\ R)(e_2 : f \equiv_{e_1} f')(e_3 : \texttt{c}\ A \equiv_{\texttt{P}\ e_1} \texttt{c}\ R)\\
&\quad\simeq (f : F\ A)(R : \texttt{Set})(f' : F\ R)(e_3' : A \equiv_{\texttt{Set}} R)(e_2 : f \equiv_{F\ e_3'} f')\\
&\quad\simeq (f : F\ A)(f' : F\ A)(e_2 : f \equiv_{F\ A} f')\\
&\quad\simeq (f : F\ A)
\end{aligned} \tag{3.25}$$

At each point during the unification process, there is only one valid way to proceed. At the first step, the second equation $f \equiv_{e_1} f'$ cannot be solved right away as the type is heterogeneous and the solution rule only applies to homogeneous equations. The first equation cannot be solved either as this would require injectivity of the functor F. The only possibility is to apply the injectivity of `c` to the third equation. At the second step, $f \equiv_{F\ e_3'} f'$ cannot be solved because the type $F\ e_3'$ is heterogeneous, so e_3' has to be solved first instead.

3.2.3 Rules for record types

One of the big advantages of having a general notion of 'unification rule' and 'most general unifier' is that we have an easy way to check the correctness of new unification rules. Alternatively it can be used to assess the impact of adding a new unification rule to the algorithm. In this section, we extend our algorithm with two unification rules that deal with η-equality for record types.

A *record type* is a type for grouping values together. One of the properties that sets a record type apart from a regular datatype with a single constructor, are the additional laws for equality of records called η-laws (not to be confused with the η-law for functions).

Definition 3.35 (Record type)**.** A *record type* $\mathtt{R} : \mathtt{Set}_\ell$ is defined by a number of *fields* (also called *projections*):

$$\begin{aligned} \mathtt{f_1} &: (r : R) \to A_1 \\ \mathtt{f_2} &: (r : R) \to A_2\ (\mathtt{f_1}\ r) \\ &\vdots \\ \mathtt{f_n} &: (r : R) \to A_n\ (\mathtt{f_1}\ r)\ \dots\ (\mathtt{f_{n-1}}\ r) \end{aligned} \tag{3.26}$$

To construct an element of the record type from values $x_1 : A_1, \dots, x_n : A_n\ x_1\ \dots\ x_{n-1}$, we use the syntax $\mathtt{record}\{\mathtt{f_1} = x_1; \dots; \mathtt{f_n} = x_n\}$. Applying one of the projections to a record constructed this way gives back the field:

$$\mathtt{f_i}\ (\mathtt{record}\{\mathtt{f_1} = x_1; \dots; \mathtt{f_n} = x_n\}) = x_i \tag{3.27}$$

The type A_i of each field can depend on the values of the previous fields $\mathtt{f_j}\ r$ for $j < i$. For example, $\Sigma_{x:A}\ (B\ x)$ can be defined as a record with two projections $\mathtt{fst} : \Sigma_{x:A}\ (B\ x) \to A$ and $\mathtt{snd} : (p : \Sigma_{x:A}\ (B\ x)) \to B\ (\mathtt{fst}\ p)$. Then x, y is shorthand for $\mathtt{record}\{\mathtt{fst} = x; \mathtt{snd} = y\}$.

The η-law states that for any $r : R$, we have

$$r = \mathtt{record}\{\mathtt{f_1} = \mathtt{f_1}\ r; \dots; \mathtt{f_n} = \mathtt{f_n}\ r\} \tag{3.28}$$

We use the η-law to construct two unification rules. The first rule applies η to expand a variable of record type into its constituent fields, while the second rule performs a similar expansion on an equation between two elements of a record type.[1]

Lemma 3.36 ($\eta\mathtt{var_R}$)**.** *Let* $\mathtt{R} : \mathtt{Set}_\ell$ *be a record type with fields given by* (3.26)*. Then we have an equivalence:*

$$\eta\mathtt{var_R} : (r : \mathtt{R}) \simeq (f_1 : A_1) \dots (f_n : A_n\ f_1\ \dots\ f_{n-1}) \tag{3.29}$$

Construction of $\eta\mathtt{var_R}$. We define $\eta\mathtt{var}\ r$ by $\mathtt{f_1}\ r; \dots; \mathtt{f_n}\ r$, and $\eta\mathtt{var}^{-1}\ f_1\ \dots\ f_n$ by $\mathtt{record}\{\mathtt{f_1} = f_1; \dots; \mathtt{f_n} = f_n\}$. The proofs of both `isLinv` and `isRinv` is `refl`: in the former case this is type-correct because of the η-law (3.28), and in the latter case because of the computation rules for projections (3.27). □

[1] A cubical type theorist might say these are two instances of the same rule.

Example 3.37. This rule is especially useful for solving equations where one side is a projection applied to a variable. Consider the type $A \times B = \Sigma_{\_:A} B$. Then we can solve the equation $\mathtt{fst}\ p \equiv_{\mathbb{N}} \mathtt{zero}$ as follows:

$$\begin{aligned}
&(\underline{p} : \mathbb{N} \times \mathbb{N})(e : \mathtt{fst}\ p \equiv_{\mathbb{N}} \mathtt{zero}) \\
&\quad \simeq (\underline{x} : \mathbb{N})(y : \mathbb{N})(\underline{e} : x \equiv_{\mathbb{N}} \mathtt{zero}) \\
&\quad \simeq (y : \mathbb{N})
\end{aligned} \tag{3.30}$$

Lemma 3.38 ($\eta\mathtt{eq}$)**.** *Let* $\mathtt{R} : \mathtt{Set}_\ell$ *be a record type with fields given by* (3.26)*. Then we have an equivalence:*

$$\eta\mathtt{eq} : (e : r \equiv_R s) \simeq (e_1 : \mathtt{f_1}\ r \equiv_{A_1} \mathtt{f_1}\ s) \dots (e_n : \mathtt{f_n}\ r \equiv_{A_n\ e_1\ \dots\ e_{n-1}} \mathtt{f_n}\ s) \tag{3.31}$$

Construction of $\eta\mathtt{eq}$. To construct $\eta\mathtt{eq}$, we rely on $\eta\mathtt{var}$ and $\mathtt{cong}$: we define $\eta\mathtt{eq}\ e = \mathtt{cong}\ \eta\mathtt{var}\ e$ and $\eta\mathtt{eq}^{-1}\ \bar{e} = \overline{\mathtt{cong}}\ \eta\mathtt{var}^{-1}\ \bar{e}$. The proofs of $\mathtt{isLinv}$ and $\mathtt{isRinv}$ are straightforward applications of $\overline{\mathtt{J}}$. □

Example 3.39. This rule is useful when one side of an equation is of the form $\mathtt{record}\{\dots\}$. If $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, then we can solve the equation $x, y \equiv_{\mathbb{N}\times\mathbb{N}} f\ z$ as follows:

$$\begin{aligned}
&(x\ y\ z : \mathbb{N})(\underline{e} : x, y \equiv_{\mathbb{N}\times\mathbb{N}} f\ z) \\
&\quad \simeq (\underline{x}\ y\ z : \mathbb{N})(\underline{e_1} : x \equiv_{\mathbb{N}} \mathtt{fst}\ (f\ z))(e_2 : y \equiv_{\mathbb{N}} \mathtt{snd}\ (f\ z)) \\
&\quad \simeq (\underline{y}\ z : \mathbb{N})(\underline{e_2} : y \equiv_{\mathbb{N}} \mathtt{snd}\ (f\ z)) \\
&\quad \simeq (z : \mathbb{N})
\end{aligned} \tag{3.32}$$

3.3 Computational behaviour of unification rules

Until now, we have only been interested in an equivalence representing a unification rule insofar that it has the correct type. But as a term in type theory, it also has a certain computational behaviour. This computational behaviour is important when we use that term as a component in some larger construction, for example in the translation of pattern matching to eliminators in the next chapter.

In particular, an important step during the translation of a case split to the application of an eliminator is to generate auxiliary equations (Definition 4.22) that are then solved by unification (Definition 4.24). However, in the end these equations are filled in with $\overline{\mathtt{refl}}$ (see in particular Definition 4.22).

Example 3.40. Consider the definition of the function `tail` (Example 1.10):

$$\begin{aligned} &\texttt{tail} : (n : \mathbb{N}) \to \texttt{Vec}\ A\ (\texttt{suc}\ n) \to \texttt{Vec}\ A\ n \\ &\texttt{tail}\ .m\ (\texttt{cons}\ m\ x\ xs) \ = \ xs \end{aligned} \tag{3.33}$$

To translate this definition to eliminators, the first step is to generalize the problem to constructing $\texttt{tail'} : (k\ n : \mathbb{N}) \to \texttt{Vec}\ A\ k \to k \equiv_{\mathbb{N}} \texttt{suc}\ n \to \texttt{Vec}\ A\ n$ and then let $\texttt{tail}\ n\ xs = \texttt{tail'}\ (\texttt{suc}\ n)\ n\ xs\ \texttt{refl}$. Note in particular that the final argument to `tail'` is `refl`!

The next step in the translation is to apply case analysis (Lemma 4.5) to xs, which instantiates k with `zero` in the case for `nil` and `suc` m in the case for `cons`. The equations $\texttt{zero} \equiv_{\mathbb{N}} \texttt{suc}\ n$ and $\texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n$ are then solved by unification:

- In the first case, we get a negative unifier $f_1 : (n : \mathbb{N})(\texttt{zero} \equiv_{\mathbb{N}} \texttt{suc}\ n) \simeq \bot$ and in particular $f\ n\ e : \bot$, so the case can be handled by applying $\texttt{absurd} : (A : \texttt{Set}_\ell) \to \bot \to A$ (see Example 1.7).
- In the second case, we get a positive unifier $f_2 : (m\ n : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ m)(\texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n) \simeq (m : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ m)$. This unifier is used in the further translation of the right-hand side of `tail`.

When (the translated version of) `tail` is called with arguments m and $\texttt{cons}\ m\ x\ xs$, it will evaluate to $\texttt{tail'}\ (\texttt{suc}\ m)\ m\ (\texttt{cons}\ m\ x\ xs)\ \texttt{refl}$ by definition. In particular, the proof of $\texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n$ that is passed to the equivalence f_2 is `refl`.

So if we care about the computational behaviour of the output of this translation, we should worry about what happens when we apply a unification rule to $\overline{\texttt{refl}}$, i.e. when the equations on one side of the equivalence hold in fact definitionally.

Intuitively, a unification rule $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$ should satisfy the property that if the equations on the left hold definitionally then the ones on the right also hold definitionally and vice versa. Moreover, the proofs $\texttt{isLinv}\ r$ and $\texttt{isRinv}\ r$ should be trivial in those cases. In other words, the various components of the equivalence should satisfy the principle '`refl` in, `refl` out'. This leads us to the following definition of a strong unification rule:

Definition 3.41 (Strong unification rule)**.** A positive unification rule $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$ is a *strong unification rule* if for any Γ_0 and for any $\bar{s}$ and $\bar{s}'$ such that $\Gamma_0 \vdash \overline{\texttt{refl}} : \bar{u}[\Gamma \mapsto \bar{s}] \equiv_\Delta \bar{v}[\Gamma \mapsto \bar{s}]$ and $\Gamma_0 \vdash \overline{\texttt{refl}} : \bar{u}'[\Gamma' \mapsto \bar{s}'] \equiv_{\Delta'} \bar{v}'[\Gamma' \mapsto \bar{s}']$, it satisfies the following five properties:

1. $\Gamma_0 \vdash f\ \bar{s}\ \overline{\texttt{refl}} = (\bar{s}'; \overline{\texttt{refl}}) : \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$ for some $\bar{s}'$.

2. $\Gamma_0 \vdash \mathtt{linv}\ r\ \bar{s}'\ \overline{\mathtt{refl}} = (\bar{s}_1; \overline{\mathtt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ for some $\bar{s}_1$.
3. $\Gamma_0 \vdash \mathtt{rinv}\ r\ \bar{s}'\ \overline{\mathtt{refl}} = (\bar{s}_2; \overline{\mathtt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ for some $\bar{s}_2$.
4. $\Gamma_0 \vdash \mathtt{isLinv}\ r\ \bar{s}\ \overline{\mathtt{refl}} = \overline{\mathtt{refl}} : \mathtt{linv}\ f\ (f\ \bar{s}\ \overline{\mathtt{refl}}) \equiv_{\Gamma(\bar{e}:\bar{u}\equiv_\Delta \bar{v})} (\bar{s}; \overline{\mathtt{refl}})$.
5. $\Gamma_0 \vdash \mathtt{isRinv}\ r\ \bar{s}'\ \overline{\mathtt{refl}} = \overline{\mathtt{refl}} : f\ (\mathtt{rinv}\ f\ \bar{s}'\ \overline{\mathtt{refl}}) \equiv_{\Gamma'(\bar{e}':\bar{u}'\equiv_{\Delta'}\bar{v}')} (\bar{s}'; \overline{\mathtt{refl}})$.

The trivial equivalence $\mathtt{id} : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ is clearly a strong unification rule, and we can compose strong unification rules:

Lemma 3.42. *If r and r' are strong unification rules, then $r \circ r'$ is one as well.*

Proof. This follows directly from the definition of a strong unification rule and the composition of two equivalences. □

Most of the unification rules we have seen up until now are strong:

Lemma 3.43. $\mathtt{solution}$ *and* $\mathtt{injectivity}_\mathtt{c}$ *are strong unification rules.*

Proof. This follows directly from the construction of these rules (see Lemma 3.21 for $\mathtt{solution}$, and Lemma 4.16 for $\mathtt{injectivity}_\mathtt{c}$.). □

For the deletion rule (3.22), the computational behaviour depends on the type of the equation being eliminated and the construction of the proof of K. In a theory with a general K rule, the deletion rule is also a strong unification rule. However, if we construct K for a specific datatype such as $\mathbb{N}$ from the basic eliminator, then the resulting unification rule won't be strong as evaluation gets stuck in case the left- and right-hand side of the equation are not of the form $\mathtt{zero}$ or $\mathtt{suc}\ m$.

Lemma 3.44. $\eta\mathtt{var}$ *and* $\eta\mathtt{eq}$ *are strong unification rules.*

Proof. For $\eta\mathtt{var}$, the first three properties are trivial as this rule does not involve equations, and the last two properties holds as well since $\mathtt{isLinv}\ \eta\mathtt{var}\ r$ and $\mathtt{isRinv}\ \eta\mathtt{var}\ r$ are equal to $\mathtt{refl}$ by definition.

For $\eta\mathtt{eq}$, the functions $\eta\mathtt{eq}$, $\eta\mathtt{eq}^{-1}$, $\mathtt{isLinv}\ \eta\mathtt{eq}$ and $\mathtt{isRinv}\ \eta\mathtt{eq}$ all map $\overline{\mathtt{refl}}$ to $\overline{\mathtt{refl}}$ by definition of $\overline{\mathtt{cong}}$ and $\overline{\mathrm{J}}$, so it is also trivially a strong rule. □

In the special case of a most general unifier, the telescope Δ' on the right becomes trivial so we can give a simpler definition of strongness. In particular, the first property always holds so we may omit it, and for the second, third, and fifth property it is sufficient to require that they hold in the case that $\bar{s}'$ is a list of variables Γ' (since there are no equations in Δ' that should hold as a precondition). This leads us to the following definition of a strong unifier:

Definition 3.45 (Strong unifier)**.** A most general unifier $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ is *strong* if it satisfies the following four properties:

1. $\Gamma' \vdash \mathtt{linv}\ f\ \Gamma' = (\bar{s}_1; \overline{\mathtt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ for some $\bar{s}_1$.
2. $\Gamma' \vdash \mathtt{rinv}\ f\ \Gamma' = (\bar{s}_2; \overline{\mathtt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ for some $\bar{s}_2$.
3. For any Γ_0 and $\bar{s}$ such that $\Gamma_0 \vdash \overline{\mathtt{refl}} : \bar{u}[\Gamma \mapsto \bar{s}] \equiv_\Delta \bar{v}[\Gamma \mapsto \bar{s}]$, we have $\Gamma_0 \vdash \mathtt{isLinv}\ f\ \bar{s}\ \overline{\mathtt{refl}} = \mathtt{refl} : \mathtt{linv}\ f\ (f\ \bar{s}\ \overline{\mathtt{refl}}) \equiv_{\Gamma(\bar{e}:\bar{u}\equiv_\Delta\bar{v})} (\bar{s}; \overline{\mathtt{refl}})$.
4. $\Gamma' \vdash \mathtt{isRinv}\ f\ \Gamma' = \overline{\mathtt{refl}} : f\ (\mathtt{rinv}\ f\ \Gamma') \equiv_{\Gamma'} \Gamma'$.

From the first two properties we deduce in particular that the equations $\bar{u} \equiv_\Delta \bar{v}$ are indeed satisfied definitionally under the substitution embedded in the most general unifier f.

Lemma 3.46. *If $f = r_1 \circ r_2 \circ \ldots \circ r_n : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ is composed of strong unification rules $r_1, r_2, \ldots, r_n$, then f is a strong unifier.*

Proof. Since a strong unifier is a special case of a strong unification rule, this follows directly from Lemma 3.42. □

Lemma 3.47. *If $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ is a strong unifier, then $\Gamma' \vdash \mathtt{linv}\ f\ \Gamma' = \mathtt{rinv}\ f\ \Gamma' : \Gamma$.*

Proof. By the second property of a strong unifier, we have $\bar{s}_2$ such that

$$\Gamma' \vdash \mathtt{rinv}\ f\ \Gamma' = (\bar{s}_2; \overline{\mathtt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{3.34}$$

In particular, $(\bar{s}_2; \overline{\mathtt{refl}})$ has type $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$, so we know that

$$\Gamma' \vdash \overline{\mathtt{refl}} : \bar{u}[\Gamma \mapsto \bar{s}_2] \equiv_\Delta \bar{v}[\Gamma \mapsto \bar{s}_2] \tag{3.35}$$

By the third property of a strong unifier, this implies that

$$\Gamma' \vdash \mathtt{isLinv}\ f\ \bar{s}_2\ \overline{\mathtt{refl}} = \overline{\mathtt{refl}} : (\bar{s}_2; \overline{\mathtt{refl}}) \equiv_{\Gamma(\bar{e}:\bar{u}\equiv_\Delta\bar{v})} (\bar{s}_2; \overline{\mathtt{refl}}) \tag{3.36}$$

Since the left- and right-hand side of a definitional equality always have definitionally equal types, it follows in particular that the type of $\texttt{isLinv}\ f\ \bar{s}_2\ \overline{\texttt{refl}}$ must be definitionally equal to the type of $\texttt{refl}$, i.e.

$$\Gamma' \vdash \texttt{linv}\ f\ (f\ \bar{s}_2\ \overline{\texttt{refl}}) = (\bar{s}_2; \overline{\texttt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{3.37}$$

By similar reasoning, the fourth property gives us that the type of $\texttt{isRinv}\ f\ \Gamma'$ must be definitionally equal to that of $\overline{\texttt{refl}}$, so in particular

$$\Gamma' \vdash f\ (\texttt{rinv}\ f\ \Gamma') = \Gamma' : \Gamma' \tag{3.38}$$

But $\texttt{rinv}\ f\ \Gamma' = \bar{s}_2; \overline{\texttt{refl}}$, so we also have

$$\Gamma' \vdash f\ (\bar{s}_2; \overline{\texttt{refl}}) = \Gamma' : \Gamma' \tag{3.39}$$

Applying $\texttt{linv}\ f$ to both sides of this equations gives us that

$$\Gamma' \vdash \texttt{linv}\ f\ (f\ (\bar{s}_2; \overline{\texttt{refl}})) = \texttt{linv}\ f\ \Gamma' : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{3.40}$$

Putting this together with (3.37) gives us that

$$\Gamma' \vdash \texttt{linv}\ f\ \Gamma' = (\bar{s}_2; \overline{\texttt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{3.41}$$

Since $\texttt{rinv}\ f\ \Gamma' = (\bar{s}_2; \overline{\texttt{refl}})$, this gives us that $\Gamma' \vdash \texttt{linv}\ f\ \Gamma' = \texttt{rinv}\ f\ \Gamma' : \Gamma$, as we wanted to prove. □

This lemma implies that we can write f^{-1} for both $\texttt{linv}\ f$ and $\texttt{rinv}\ f$ when f is a strong unifier.

Discussion about the definition of a strong unifier. There are other possible definitions of a strong unifier. In particular, to guarantee the good computational properties of functions constructed through specialization by unification (Lemma 4.26), we only need properties 1, 3, and a weaker version of property 4. In our previous work (Cockx et al., 2016a) we used an even weaker definition of a strong unifier:

Definition 3.48 (DEPRECATED, version from Cockx et al. (2016a))**.** A most general unifier $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ is *strong* if for any $\bar{x}' : \Gamma'$, it satisfies the following two properties:

- $f\ (f^{-1}\ \bar{x}') = \bar{x}'$
- $\texttt{isLinv}\ f\ (f^{-1}\ \bar{x}') = \overline{\texttt{refl}}$

This definition ensures exactly the properties needed for Lemma 4.26 to hold. However, when writing down the proof of Theorem 4.29 in detail we discovered it required an additional property, namely that $f^{-1}\ \bar{x}'$ must be definitionally equal to something of the form $\bar{s}; \overline{\texttt{refl}}$. Additionally, in various places we relied implicitly on the fact that $\texttt{linv}\ f$ and $\texttt{rinv}\ f$ should be definitionally equal. These discoveries lead to our current definition of a strong unifier.

3.4 Higher-dimensional unification

When constructing the indexed versions of the injectivity, conflict, and cycle rules (Section 3.2.2), we required that the telescope of the equations on the left-hand side should be exactly $\overline{\texttt{D}} = (\bar{u} : \Xi)(x : \texttt{D}\ \bar{u})$. This means these rules can only be applied to an equation where the type is *fully general*, i.e. a datatype applied to distinct equality proofs for its indices. This is convenient when the equations we start with are of this form because it allows us to simplify all equations at the same time.

The main question posed in this section is what we can do if we encounter an equation of the form $\texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{v}} \texttt{c}\ \bar{v}$ but the indices $\bar{v}$ are not fully general.

Example 3.49. Suppose the unification algorithm is trying to solve an equation

$$(e : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)} \texttt{cons}\ n\ y\ ys) \tag{3.42}$$

of type $\texttt{Vec}\ A\ (\texttt{suc}\ n)$ where n is a regular variable rather than an equality proof. In this case it is not possible to apply the $\texttt{injectivity}_{\texttt{cons}}$ rule directly.

There is no fundamental reason why unification should fail on this example. On the other hand, always applying the injectivity rule even when the indices are not fully general is unsound (Example 3.3). This is not just a theoretical problem either: see for example issues #1411 and #1775 on the Agda bug tracker (Abel, 2015b; Sicard-Ramírez, 2016).

In previous work, we tried different approaches to solve this problem that worked in some cases but were ultimately unsatisfactory. In Cockx et al. (2014b) we restricted all unification rules to homogeneous equations and additionally imposed a *self-unifiability criterion* to the indices of the datatype when applying the injectivity rule. In practice, this meant that the injectivity rule could only be applied when the indices consisted of closed constructor forms only (e.g. $\texttt{suc}\ (\texttt{suc}\ \texttt{zero})$), but not $\texttt{suc}\ n$), a severe restriction to the applicability of the rule. In Cockx et al. (2016a) we used the general (heterogeneous) version of the injectivity rule and relied on *reverse unification* to generalize the indices.

This method had some potential in theory, but turned out to be too difficult to implement in practice. Neither did we take into account the type of the constructor in question, so we were unable to include useful heuristics such as forced constructor arguments (Brady et al., 2003).

In this section, we describe a general technique for solving equations between constructors of indexed datatypes. First, we study why the problem is so difficult by looking at the analogous problem for the conflict and cycle rules, and make a first attempt at generalizing the injectivity rule (Section 3.4.1). We continue to show how to generalize the equality proofs in the indices in the general case by introducing new equations between equality proofs (Section 3.4.2). Borrowing terminology from homotopy type theory, we call them *higher-dimensional equations*. To solve these higher-dimensional equations, we show how to lift existing unification rules to higher dimensions (Section 3.4.3).

3.4.1 Generalizing unification rules

Before we try to tackle the problem of how to apply the injectivity rule on an equation when the indices are not fully general, we first consider the analogous problem for the conflict and cycle rules. The reason to take on these rules first is because they shed some light on why the problem is harder for the injectivity rule.

Lemma 3.50 (Generalized conflict)**.** *Consider a unification problem of the form* $(\bar{s}_1; \mathtt{c}_1\ \bar{t}_1) \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} (\bar{s}_2; \mathtt{c}_2\ \bar{t}_2)$ *where* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ *is a datatype and* $\mathtt{c}_1 : \Delta_1 \to \mathtt{D}\ \bar{u}_1$ *and* $\mathtt{c}_2 : \Delta_2 \to \mathtt{D}\ \bar{u}_2$ *are two distinct constructors of* $\mathtt{D}$*. Then we have an equivalence*

$$\mathtt{conflict'}_{\mathtt{c}_1,\mathtt{c}_2} : \big((\bar{s}_1; \mathtt{c}_1\ \bar{t}_1) \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} (\bar{s}_2; \mathtt{c}_2\ \bar{t}_2)\big) \simeq \bot \tag{3.43}$$

The indices $\bar{v}$ are arbitrary, i.e. they don't have to be variables like in the standard conflict rule (Lemma 3.29). However, the type of the final equation still has to be the datatype $\mathtt{D}$ applied to these indices. In particular it cannot be a variable itself, or else we would run into the problem described in Example 3.33.

Before we give the proof of this lemma, we first want to show how a naive proof attempt fails. It goes as follows: to construct a function $\big((\bar{s}_1; \mathtt{c}_1\ \bar{t}_1) \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} (\bar{s}_2; \mathtt{c}_2\ \bar{t}_2)\big) \to \bot$, it suffices (by the $\overline{\mathrm{J}}$ rule) to construct a function $\mathtt{c}_1\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}} \mathtt{c}_2\ \bar{t}_2 \to \bot$. This function is constructed by calling the indexed conflict rule (3.29) with $\overline{\mathtt{refl}}$ for the proof of $\bar{u}_1[\Delta_1 \mapsto \bar{t}_1] \equiv_\Xi \bar{u}_2[\Delta_2 \mapsto \bar{t}_2]$. Since any function to $\bot$ is an equivalence, we are done.

Think a moment about what is wrong with this proof. It uses the $\overline{\mathrm{J}}$ rule to eliminate the equations $\bar{s}_1 \equiv_\Phi \bar{s}_2$, but there is no guarantee that $\bar{s}_1$ or $\bar{s}_2$ are

in fact variables. Moreover, their structure as a term may be important for satisfying the assumptions of the lemma, so simply generalizing the statement of the lemma is not possible. In other words, the error in this proof attempt stems from a confusion about the status of $\bar{s}_1$ and $\bar{s}_2$ as variables at the meta-level, while they can be arbitrary terms at the object level!

We work around this issue by using the following lemma:

Lemma 3.51. *Let* $f : A \to B$ *and* $P : B \to \mathtt{Set}$ *and* $e : s \equiv_A t$ *and* $u : P\ (f\ s)$ *and* $v : P\ (f\ t)$*. Then the types* $u \equiv_{P\ (f\ e)} v$ *and* $u \equiv_{P\ (\mathtt{cong}\ f\ e)} v$ *are equivalent.*

Proof. Notice the rather subtle difference between these two types: the first one expands to $\mathtt{subst}\ (P \circ f)\ e\ u \equiv_{P\ (f\ t)} v$, while the second one expands to $\mathtt{subst}\ P\ (\mathtt{cong}\ f\ e)\ u \equiv_{P\ (f\ t)} v$. To prove that they are equivalent, it is sufficient to prove that $\mathtt{subst}\ (P \circ f)\ e\ u \equiv_{P\ (f\ t)} \mathtt{subst}\ P\ (\mathtt{cong}\ f\ e)\ u$. But this follows directly by eliminating e using J. □

Construction of $\mathtt{conflict'}_{\mathtt{c}_1,\mathtt{c}_2}$. We start by expanding the definition of telescopic equality: we have to derive an element of $\bot$ from

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}_1\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}[\Phi \mapsto \bar{e}_1]} \mathtt{c}_2\ \bar{t}_2) \tag{3.44}$$

By Lemma 3.51, the type of e_2 is equivalent to $\mathtt{c}_1\ \bar{t}_1 \equiv_{\mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\Phi.\ \bar{v})\ \bar{e})} \mathtt{c}_2\ \bar{t}_2$. So we call the conflict rule (3.29) with arguments $((\overline{\mathtt{cong}}\ (\lambda\Phi.\ \bar{v})\ \bar{e}_1); e_2)$ to get an element of type $\bot$. Since any function to $\bot$ is an equivalence, this finishes the proof. □

Similarly, we can generalize the cycle rule:

Lemma 3.52 (Generalized cycle)**.** *Consider a unification problem of the form* $(\bar{s}_1; t_1) \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} (\bar{s}_2; t_2)$ *where* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ *is a datatype and* $t_1 \prec t_2$*. Then we have an equivalence*

$$\mathtt{cycle'}_{t_1,t_2} : \big((\bar{s}_1; t_1) \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} (\bar{s}_2; t_2)\big) \simeq \bot \tag{3.45}$$

Construction of $\mathtt{cycle'}_{t_1,t_2}$. Analogously to the construction of $\mathtt{conflict'}_{\mathtt{c}_1,\mathtt{c}_2}$. □

For injectivity, it isn't as easy to generalize the rule to arbitrary indices like we just did for conflict and cycle. The problem here is harder because we also have to construct an inverse function and prove that it is indeed a left and right inverse, while this was trivial for the two negative rules. In the special case where the index telescope Ξ satisfies the K rule, we can construct the generalization:

Lemma 3.53 (Generalized injectivity)**.** *Consider a unification problem of the form* $(\bar{s}_1; \mathtt{c}\ \bar{t}_1) \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} (\bar{s}_2; \mathtt{c}\ \bar{t}_2)$ *where* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ *is a datatype with (at least) one constructor* $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$*, and assume* Ξ *satisfies* K*, i.e. we have* $\mathtt{deletion}_\Xi : (\bar{e} : \bar{w} \equiv_\Xi \bar{w}) \simeq ()$ *for all* $\bar{w} : \Xi$*. Then we have an equivalence*

$$\mathtt{injectivity'_c} : \big((\bar{s}_1; \mathtt{c}\ \bar{t}_1) \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} (\bar{s}_2; \mathtt{c}\ \bar{t}_2)\big) \simeq ((\bar{s}_1; \bar{t}_1) \equiv_{\Phi\Delta} (\bar{s}_2; \bar{t}_2)) \quad (3.46)$$

In case Φ is the empty telescope, this generalized injectivity rule is similar to the specialized injectivity rule from Cockx et al. (2014b), but here we ask that the types of the indices Ξ satisfy K, instead of asking that the indices $\bar{u}$ are self-unifiable.

Construction of $\mathtt{injectivity'_c}$. As for the previous lemma, we expand the definition of telescopic equality and apply Lemma 3.51 to get to

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\Phi.\,\bar{v})\ \bar{e}_1)}\ \mathtt{c}\ \bar{t}_2) \quad (3.47)$$

Since Ξ satisfies K, it follows that $(e'_1 : \bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2])$ is equivalent to $()$. So the previous telescope is equivalent to

$$\begin{array}{l}(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_1 : \bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2]) \\ (e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\Phi.\,\bar{v})\ \bar{e}_1)}\ \mathtt{c}\ \bar{t}_2)\end{array} \quad (3.48)$$

Again by K, we have that the proofs $\overline{\mathtt{cong}}\ (\lambda\Phi.\,\bar{v})\ \bar{e}_1$ and $\bar{e}'_1$ of type $\bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2]$ are equal. This means the previous telescope is equivalent to

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_1 : \bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2])(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{e}'_1}\ \mathtt{c}\ \bar{t}_2) \quad (3.49)$$

Finally, we can apply the injectivity rule (3.28) to prove that the part of the telescope containing $\bar{e}'_1$ and e_2 is equivalent to $\bar{t}_1 \equiv_\Delta \bar{t}_2$, so the previous telescope is equivalent to

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}_2 : \bar{t}_1 \equiv_\Delta \bar{t}_2) \quad (3.50)$$

which is what we wanted to prove. □

Like for the deletion rule, this generalized injectivity rule usually won't be a strong rule because its computational behaviour depends on the construction of the proof of K for the index types.

3.4.2 A generalized injectivity rule

The generalized injectivity rule from the previous section is unsatisfactory because it requires the index types of the datatype to satisfy K. This means we

didn't actually solve the problem of depending on K yet, we only moved it to the indices. However, the proof taught us something about how to solve the problem in general: it introduced new equality proofs $\bar{e}'_1$ and used the K rule to substitute these for the indices of $\mathtt{D}$, allowing us to apply the injectivity rule. In other words, it moved the problem from talking about equalities between *terms* to equalities between *equality proofs*.

In this section, we show how to apply this idea in a more general way to remove the dependency on K completely. We do this by applying—what else—unification to the equations between the indices of the datatype. Since the indices in the type of an equation can depend on the equality proofs of the previous equations, this means we have to solve not just equalities between terms but also equalities between other equality proofs, i.e. higher-dimensional equations.

At first sight, it would seem that an entirely new set of unification rules is needed to solve higher-dimensional equations (except for the solution rule, which can be used at any dimension). However, it is possible to reuse the existing unification rules on higher-dimensional problems. For example, the $\mathtt{injectivity}_{\mathtt{suc}}$ rule can be used not just to simplify equations of the form $\mathtt{suc}\ x \equiv_{\mathbb{N}} \mathtt{suc}\ y$ to $x \equiv_{\mathbb{N}} y$, but also $\mathtt{cong}\ \mathtt{suc}\ e_1 \equiv_{\mathtt{suc}\ x \equiv_{\mathbb{N}} \mathtt{suc}\ y} \mathtt{cong}\ \mathtt{suc}\ e_2$ to $e_1 \equiv_{x \equiv_{\mathbb{N}} y} e_2$.

In general, whenever the unification algorithm encounters a higher-dimensional unification problem of the form $\overline{\mathtt{cong}}\ (\lambda\Phi.\ \bar{u})\ \bar{e} \equiv_{\bar{s} \equiv_{\Delta} \bar{t}} \overline{\mathtt{cong}}\ (\lambda\Phi.\ \bar{v})\ \bar{e}$, it first considers the simpler unification problem $\bar{u} \equiv_{\Delta} \bar{v}$. If it manages to find a solution to this one-dimensional problem, it can then *lift* this solution to get a solution to the original problem. The technical result that makes this possible is Lemma 3.60 in the next section.

Let's first take a look of how this works on an example.

Example 3.54. Consider the unification problem:

$$\Gamma(e : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys) \tag{3.51}$$

where $\Gamma = (n : \mathbb{N})(x\ y : A)(xs\ ys : \mathtt{Vec}\ A\ n)$. The $\mathtt{injectivity}_{\mathtt{cons}}$ rule cannot be applied, as the index $\mathtt{suc}\ n$ is not fully general (i.e. it is not an equation variable). Instead, we solve this unification problem in three steps: in the first step, we generalize over the indices in order to apply the injectivity rule, generating higher-dimensional equations in the process. In the second step, we bring down these equations by one dimension so we can solve them by applying known unification rules. Finally, we lift the one-dimensional unifier to the higher-dimensional problem.

Step 1: generalizing the indices. We generalize the problem by introducing an extra equation $e_1 : \mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n$ to the telescope, together with a proof p that e_1 is equal to $\mathtt{refl}$:

$$\begin{aligned}&\Gamma(e : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys)\\ &\quad\simeq \Gamma(e_1 : \mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n)(e_2 : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys)\\ &\qquad (p : e_1 \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})\end{aligned} \tag{3.52}$$

This is nothing but an application of the $\mathtt{solution}$ rule in the reverse direction, as applying $\mathtt{solution}$ to p would bring us back to the first equation.[2]

Since the index in the type of e_2 is now fully general, we are free to apply the $\mathtt{injectivity}_{\mathtt{cons}}$ rule:

$$\begin{aligned}&\Gamma(\underline{e_1} : \mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n)(\underline{e_2} : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys)\\ &(p : e_1 \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})\\ &\quad\simeq \Gamma(e'_1 : n \equiv_{\mathbb{N}} n)(e'_2 : x \equiv_A y)(e'_3 : xs \equiv_{\mathtt{Vec}\ A\ e'_1} ys)\\ &\qquad (p : \mathtt{cong}\ \mathtt{suc}\ e'_1 \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})\end{aligned} \tag{3.53}$$

Applying the injectivity rule to e_2 has instantiated the variable e_2 with $\mathtt{cong}\ \mathtt{suc}\ e'_1$. This instantiation is determined by the computational behaviour of the $\mathtt{injectivity}_{\mathtt{cons}}$ rule (Lemma 3.23). As you can see, p is a non-trivial equation between equality proofs, i.e. a *higher-dimensional equation*.

Step 2: lowering the dimension of equations. To solve the higher-dimensional equation p, we first consider a one-dimensional version of this problem:

$$(w'_1 : \mathbb{N})(w'_2 : A)(w'_3 : \mathtt{Vec}\ A\ w'_1)(p : \mathtt{suc}\ w'_1 \equiv_{\mathbb{N}} \mathtt{suc}\ n) \tag{3.54}$$

The equality proofs e'_1, e'_2 and e'_3 from (3.53) have been replaced by regular variables w'_1, w'_2 and w'_3. To reflect this change, $\mathtt{cong}\ \mathtt{suc}\ e'_1 : \mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n$ has been replaced by $\mathtt{suc}\ w'_1$ and $\mathtt{refl} : \mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n$ by $\mathtt{suc}\ n$.

Now this is a problem we know how to solve: we apply $\mathtt{injectivity}_{\mathtt{suc}}$ and $\mathtt{solution}$ to find an equivalence f between this telescope and $(w'_2 : A)(w'_3 : \mathtt{Vec}\ A\ n)$. This solves the one-dimensional problem.

[2] This is the exact same technique as used by McBride (2000): to do a case split on a variable $x : \mathtt{Vec}\ A\ m$ where m is not fully general, he introduces a new variable $n : \mathbb{N}$ together with an equality $e : m \equiv_{\mathbb{N}} n$. This means that now $x : \mathtt{Vec}\ A\ m$ where m are just variables, so it is possible to perform a case split on x. The only difference in our case is that we are working one dimension higher, i.e. we work with equations between elements of the datatype instead of elements of the datatype itself.

Step 3: lifting unifiers to a higher dimension. How does this help us with the higher-dimensional problem? By Lemma 3.60 (Section 3.4.3), we can *lift* the equivalence f to get a new equivalence $f^{\uparrow}$:

$$\begin{aligned}&(e'_1 : n \equiv_{\mathbb{N}} n)(e'_2 : x \equiv_A y)(e'_3 : xs \equiv_{\mathtt{Vec}\ A\ e'_1} ys)\\&(p : \mathtt{cong}\ \mathtt{suc}\ e'_1 \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})\\&\quad\simeq (e''_2 : x \equiv_A y)(e''_3 : xs \equiv_{\mathtt{Vec}\ A\ n} ys)\end{aligned} \tag{3.55}$$

This solves the higher-dimensional equation p, as well as the reflexive equation e'_1, without relying on the fact that $\mathbb{N}$ satisfies uniqueness of identity proofs!

Finally, we apply the `solution` rule twice to solve the equations e''_2 and e''_3. So putting everything together, we have found an equivalence between the original telescope (3.51) and $(n : \mathbb{N})(x : A)(xs : \mathtt{Vec}\ A\ n)$, solving the unification problem.

Now that we have seen how to solve the problem in an example, let's try to generalize the solution. The main result of this section is the following theorem.

Theorem 3.55. *Let* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ *be a datatype and* $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$ *be a constructor of* D*. Consider a unification problem of the form*

$$(\bar{e} : (\bar{s}_1; \mathtt{c}\ \bar{t}_1) \equiv_{\Phi(z:\mathtt{D}\ \bar{v})} (\bar{s}_2; \mathtt{c}\ \bar{t}_2)) \tag{3.56}$$

Suppose we have an equivalence $f : \Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v}) \simeq \Delta'$*. Then we also have an equivalence*

$$(\bar{e} : (\bar{s}_1; \mathtt{c}\ \bar{t}_1) \equiv_{\Phi(z:\mathtt{D}\ \bar{v})} (\bar{s}_2; \mathtt{c}\ \bar{t}_2)) \simeq (\bar{e}' : f\ \bar{s}_1\ \bar{t}_1\ \overline{\mathtt{refl}} \equiv_{\Delta'} f\ \bar{s}_2\ \bar{t}_2\ \overline{\mathtt{refl}}) \tag{3.57}$$

Moreover, if f *is a strong unification rule then so is this new equivalence.*

The computational behaviour of the unifier f suddenly becomes relevant for the type of the resulting unification problem! In particular, we need the behaviour of f applied to $\overline{\mathtt{refl}}$ to calculate the left- and right-hand sides of the new equations $\bar{e}'$.

Construction. We follow the same three steps as in Example 3.54, so if something is unclear it may help to take a look at the corresponding step in the example.

Step 1: generalizing the indices. First, we unfold the telescopic equality in (3.56) and apply Lemma 3.51 to get an equivalence with $(\bar{e}_1 : \bar{s}_1 \equiv_\Phi$

$\bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}_e}\ \mathtt{c}\ \bar{t}_2)$ where $\bar{v}_e = \overline{\mathtt{cong}}\ (\lambda\Phi.\ \bar{v})\ \bar{e}_1$. The equality proofs $\bar{v}_e$ have type $\bar{u}_1 \equiv_\Xi \bar{u}_2$ where $\bar{u}_1$ and $\bar{u}_2$ stand for $\bar{u}[\Delta \mapsto \bar{t}_1]$ and $\bar{u}[\Delta \mapsto \bar{t}_2]$ respectively. To generalize $\bar{v}_e$, we introduce new variables $\bar{\imath} : \bar{u}_1 \equiv_\Xi \bar{u}_2$ together with equalities $\bar{p} : \bar{\imath} \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e$:

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}_e}\ \mathtt{c}\ \bar{t}_2)\\ &\quad\simeq (\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{\imath} : \bar{u}_1 \equiv_\Xi \bar{u}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{\imath}}\ \mathtt{c}\ \bar{t}_2)\\ &\qquad(\bar{p} : \bar{\imath} \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\end{aligned} \tag{3.58}$$

Since $\bar{\imath}$ consists of distinct equation variables, it's now possible to apply $\mathtt{injectivity}_\mathtt{c}$ to the equation e_2. This gives us an equivalence:

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\underline{\bar{\imath}} : \bar{u}_1 \equiv_\Xi \bar{u}_2)(\underline{e_2} : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{\imath}}\ \mathtt{c}\ \bar{t}_2)(\bar{p} : \bar{\imath} \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\\ &\quad\simeq (\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_2 : \bar{t}_1 \equiv_\Delta \bar{t}_2)(\bar{p} : \bar{u}_e \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\end{aligned} \tag{3.59}$$

where $\bar{u}_e = \overline{\mathtt{cong}}\ (\lambda\Delta.\ \bar{u})\ \bar{e}'_2$.

Step 2: lowering the dimension of equations. Consider the one-dimensional version of this unification problem $\Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$, where the equality proofs $\bar{u}_e$ and $\bar{v}_e$ have been replaced by their lower-dimensional variants $\bar{u}$ and $\bar{v}$ respectively. Since this is a one-dimensional unification problem, we can apply the known unification rules from Section 3.2 to solve it. By assumption of the theorem, unification succeeds positively with most general unifier f as a result.

Step 3: lifting unifiers to a higher dimension. Now we have to lift this solution back to the higher-dimensional problem. This lifting is explained in the next subsection. Lemma 3.60 gives us a lifted equivalence $f^\uparrow$:

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_2 : \bar{t}_1 \equiv_\Delta \bar{t}_2)(\bar{p} : \bar{u}_e \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\\ &\quad\simeq (\bar{e}' : f\ \bar{s}_1\ \bar{t}_1\ \overline{\mathtt{refl}} \equiv_{\Delta'} f\ \bar{s}_2\ \bar{t}_2\ \overline{\mathtt{refl}})\end{aligned} \tag{3.60}$$

This is exactly what we need to solve the problem in (3.59).

Now we combine the equivalences in (3.58), (3.59) and (3.60) to get the final equivalence (3.57).

To see why this is a strong unification rule, note that it is the composition of four equivalences: Lemma 3.51, $\mathtt{solution}^{-1}$, $\mathtt{injectivity}_\mathtt{c}$ and $f^\uparrow$. By Lemma 3.42, it is sufficient to prove that these four equivalences are strong unification rules individually. The first two are strong by construction, and $\mathtt{injectivity}_\mathtt{c}$ is a strong unification rule by Lemma 3.43. Finally, $f^\uparrow$ is strong too by Lemma 3.61. □

This finishes the application of higher-dimensional unification to the equation e. We have solved the injectivity problem e, and there are no more higher-dimensional unification problems in the resulting equations $\bar{e}'_1$, so we can continue unification on the new problem as normal.

It is impossible for higher-dimensional unification to end in a negative success, as this would mean we are trying to solve an ill-typed equation. For example, we can never encounter a higher-dimensional conflict:

$$\overline{\texttt{cong}}\ \texttt{c}_1\ \bar{e}_1 \equiv_{???} \overline{\texttt{cong}}\ \texttt{c}_2\ \bar{u}'\ \bar{v}'\ \bar{e}_2 \tag{3.61}$$

because the left-hand side has a type of the form $\texttt{c}_1\ \bar{u} \equiv \texttt{c}_1\ \bar{v}$ while the right-hand side has type $\texttt{c}_2\ \bar{u}' \equiv \texttt{c}_2\ \bar{v}'$. Likewise, a higher-dimensional cycle would be:

$$e \equiv_{???} \overline{\texttt{cong}}\ \texttt{c}\ e \tag{3.62}$$

where the left-hand side has some type $u \equiv v$ but the right-hand side has type $\texttt{c}\ \bar{u}' \equiv \texttt{c}\ \bar{v}'$ where u and v occur in $\bar{u}'$ and $\bar{v}'$ respectively.

Now that we know how to do higher-dimensional unification in general, we can justify the second restriction to the unification algorithm in Section 2.3.1.

Corollary 3.56. *Let* $\texttt{c} : \Delta \to \texttt{D}\ \bar{u}$ *be an invertible constructor (Definition 2.30) of the datatype* $\texttt{D} : \Xi \to \texttt{Set}_\ell$ *and* $\bar{t}_1, \bar{t}_2 : \Delta$*. Then we have an equivalence*

$$(e : \texttt{c}\ \bar{t}_1 \equiv_{\texttt{D}\ \bar{v}} \texttt{c}\ \bar{t}_2) \simeq (\bar{e}' : \bar{t}_1|_{\Delta'} \equiv_{\Delta'} \bar{t}_2|_{\Delta'}) \tag{3.63}$$

where $\bar{v} = \bar{u}[\Delta \mapsto \bar{t}_1] = \bar{u}[\Delta \mapsto \bar{t}_2]$ *and* Δ' *is the telescope of non-forced arguments of* $\texttt{c}$ *with the forced arguments filled in with the corresponding values from* $\bar{t}_1$*.*

From the well-formedness of the type $\texttt{c}\ \bar{t}_1 \equiv_{\texttt{D}\ \bar{v}} \texttt{c}\ \bar{t}_2$, it follows that the forced arguments of $\bar{t}_1$ and $\bar{t}_2$ are equal, so it doesn't matter from which side we take them.

Construction. By definition of an invertible constructor, applying unification to the unification problem $\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$ ends in a positive success with result $f : \Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v}) \simeq \Delta'$ where the computational behaviour of f is to select the non-forced arguments of $\texttt{c}$ from Δ. Applying Theorem 3.55 to f gives us the desired equivalence. □

3.4.3 Lifting unifiers to higher dimensions

In the last section, we have seen how to apply higher-dimensional unification to make the injectivity rule more generally applicable. In this section, we dive

into the heart of the problem. Our core result that makes higher-dimensional unification work is Lemma 3.60, telling us exactly how to update the left- and right-hand sides of the equations when lifting a unifier.

Suppose we have a unifier that we want to lift to a higher dimension. As a first attempt, we try to apply the following theorem from The Univalent Foundations Program (2013):

Theorem 3.57. *If a function $f : A \to B$ is an equivalence and $x, y : A$, then* $\mathtt{cong}\ f : x \equiv_A y \to f\ x \equiv_B f\ y$ *is also an equivalence.*

Construction. This is Theorem 2.11.1 from The Univalent Foundations Program (2013). □

Applying this theorem to a unifier $f : \Gamma(\bar{p} : \bar{a} \equiv_\Delta \bar{b}) \simeq \Gamma'$ results in an equivalence $\mathtt{cong}\ f : (\bar{e} : (\bar{u}; \bar{r}) \equiv_{\Gamma(\bar{p}:\bar{a}\equiv_\Delta\bar{b})} (\bar{v}; \bar{s})) \simeq (\bar{e}' : f\ \bar{u}\ \bar{r} \equiv_{\Gamma'} f\ \bar{v}\ \bar{s})$, or expanding the definition of telescopic equality:

$$\mathtt{cong}\ f : (\bar{e} : \bar{u} \equiv_\Gamma \bar{v})(\bar{q} : \bar{r} \equiv_{\bar{a}_e \equiv_{\Delta_e} \bar{b}_e} \bar{s}) \simeq (\bar{e}' : f\ \bar{u}\ \bar{r} \equiv_{\Gamma'} f\ \bar{v}\ \bar{s}) \tag{3.64}$$

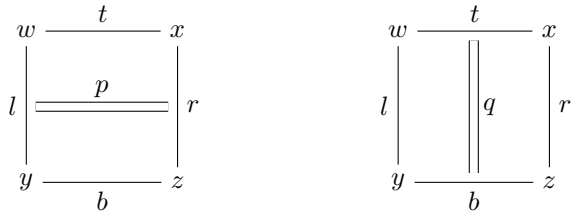
where $\bar{u}, \bar{v} : \Gamma$ and $\bar{r} : \bar{a}_u \equiv_{\Delta_u} \bar{b}_u$ and $\bar{s} : \bar{a}_v \equiv_{\Delta_v} \bar{b}_v$, and $\cdot_x$ is shorthand for $\cdot[\Gamma \mapsto \bar{x}]$. This is already *almost* what we need for higher-dimensional unification, but not quite.

To better visualize the problem, we make use of the concept of a *square*, also called a *2-path* by The Univalent Foundations Program (2013):

Definition 3.58 (Square type)**.** Let $A : \mathtt{Set}_\ell$ with terms $w, x, y, z : A$ and paths $t : w \equiv_A x$ and $b : y \equiv_A z$ and $l : w \equiv_A y$ and $r : x \equiv_A z$ between these terms. The *square type* $\mathtt{Square}\ t\ b\ l\ r$ is defined to be the dependent equality type $l \equiv_{t \equiv_A b} r$.

The type $l \equiv_{t\equiv_A b} r$ can be written a little more explicitly as $l \equiv^{(t;b)}_{\_\equiv_A\_} r$, or even more explicitly as $\overline{\mathtt{subst}}\ (\_ \equiv_A \_)\ (t; b)\ l \equiv_{x\equiv_A z} r$. If we imagine a square with top side t, bottom side b, left side l, and right side r, then $\mathtt{Square}\ t\ b\ l\ r$ can be thought of as the type of identity proofs that fill this square horizontally as visualized in Figure 3.2a.

There is a second way to construct a square type from four given points $w, x, y, z : A$ and equality proofs $t : w \equiv_A x$ and $b : y \equiv_A z$ and $l : w \equiv_A y$ and $r : x \equiv_A z$: we can 'flip' the square around its w-z axis, as illustrated by Figure 3.2b. To get to our desired result, we need to rely on the fact that both square types are in fact equivalent:



(a) Horizontal filling **(b)** Vertical filling

Figure 3.2: The `Square` type represents the possible ways to fill a square defined by four equality proofs.

Lemma 3.59 (Flipping squares)**.** *Let* $A : \texttt{Set}$ *and* $w, x, y, z : A$ *and* $t : w \equiv_A x$ *and* $b : y \equiv_A z$ *and* $l : w \equiv_A y$ *and* $r : x \equiv_A z$*. Then we have an equivalence* $\texttt{flip}\ t\ b\ l\ r : \texttt{Square}\ t\ b\ l\ r \simeq \texttt{Square}\ l\ r\ t\ b$*.*

Proof. The proof of this lemma consists completely of repeated applications of J. We start by constructing the function $\texttt{flip}\ t\ b\ l\ r : \texttt{Square}\ t\ b\ l\ r \to \texttt{Square}\ l\ r\ t\ b$. First, by J on t and b we can assume that $w = x$ and $y = z$ and both t and b are `refl`, so we are left with the goal $l \equiv_{w \equiv_A y} r \to \texttt{refl} \equiv_{l \equiv_A r} \texttt{refl}$. The identity type in the function argument has become homogeneous, so we again apply J, giving us that $l = r$ and leaving us with the goal $\texttt{refl} \equiv_{l \equiv_a l} \texttt{refl}$. Finally, one more application of J on $l : w \equiv_A y$ leaves us with the goal $\texttt{refl} \equiv_{w \equiv_A w} \texttt{refl}$, which we solve with `refl`.

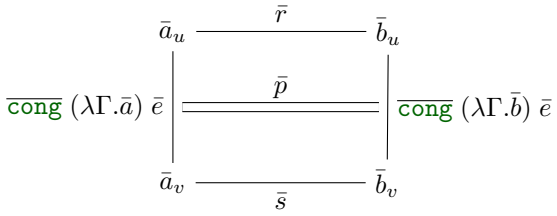
For the construction of the left and right inverse of `flip`, we just change the order of t, b, l and r in the construction of `flip`. For the proofs that they are in fact inverses, the same sequence of applications of J as used in the construction of `flip` suffices. □

Now we prove the main lemma. When we applied this lemma in the last section, we only used it for $\bar{r} = \overline{\texttt{refl}}$ and $\bar{s} = \overline{\texttt{refl}}$, but the fully general version is not harder to prove so that's what we present here.

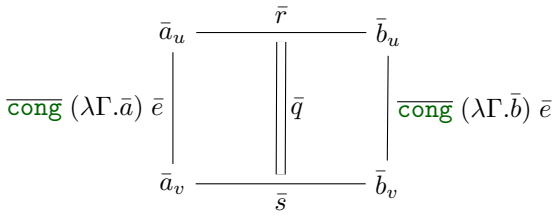
Lemma 3.60 (Lifting of unifiers)**.** *Suppose we have a unifier* $f : \Gamma(\bar{p} : \bar{a} \equiv_\Delta \bar{b}) \simeq \Gamma'$*, terms* $\bar{u}, \bar{v} : \Gamma$ *and equality proofs* $\bar{r} : \bar{a}_u \equiv_{\Delta_u} \bar{b}_u$ *and* $\bar{s} : \bar{a}_v \equiv_{\Delta_v} \bar{b}_v$*.*[3] *Then we have a* lifted unifier

$$\begin{aligned} f^\uparrow : (\bar{e} : \bar{u} \equiv_\Gamma \bar{v})(\bar{p} : \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{a})\ \bar{e} \equiv_{\bar{r} \equiv_{\Delta_e} \bar{s}} \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{b})\ \bar{e}) \\ \simeq (\bar{e}' : f\ \bar{u}\ \bar{r} \equiv_{\Gamma'} f\ \bar{v}\ \bar{s}) \end{aligned} \tag{3.65}$$

[3] We again write $\cdot_x$ for $\cdot[\Gamma \mapsto \bar{x}]$



(a) Horizontal filling $\bar{p}$



(b) Vertical filling $\bar{q}$

Figure 3.3: To construct the equivalence in Lemma 3.60, we apply Lemma 3.59 to transform the horizontal filling $\bar{p}$ into a vertical one $\bar{q}$.

Construction. By Theorem 3.57 we already have the equivalence in (3.64). By Lemma 3.51, the type of $\bar{q}$ is equivalent to $\bar{r} \equiv_{\overline{\texttt{cong}}\ (\lambda\Gamma.\bar{a})\ \bar{e}} \equiv_{\Delta_{\bar{e}}} \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{b})\ \bar{e}\ \bar{s}$. If we think of this type as a square type, then Lemma 3.59 gives us that this type is equivalent to $\overline{\texttt{cong}}\ (\lambda\Gamma.\bar{a})\ \bar{e} \equiv_{\bar{r} \equiv_{\Delta_e} \bar{s}} \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{b})\ \bar{e}$. This is illustrated in Figure 3.3. Composing this equivalence with $\texttt{cong}\ f$ gives us the desired equivalence $f^{\uparrow}$. □

Lemma 3.61. *If $f : \Gamma(\bar{p} : \bar{a} \equiv_{\Delta} \bar{b}) \simeq \Gamma'$ is a strong unifier, then so is $f^{\uparrow}$.*

Proof. $f^{\uparrow}$ is constructed as a composition of the equivalences $\texttt{cong}\ f$ (Theorem 3.57), Lemma 3.51, and `flip` (Lemma 3.59). By Lemma 3.42, we just have to verify that each of these equivalences is a strong unification rule. But this can be verified by looking at their construction (in the case of $\texttt{cong}\ f$ also using the fact that f is a strong unifier). □

3.5 Implementation

Using our framework for proof-relevant unification described in this chapter, we reimplemented the unification algorithm used by Agda for checking definitions

by dependent pattern matching. As a result, we were able to replace previous ad hoc restrictions with formally verified unification rules, fixing a number of bugs in the process. It also enabled us to add new unification rules dealing with η-equality for record types, as well as higher-dimensional unification for solving equations between constructors of indexed datatypes. Another advantage of our approach is that the implementation is now much cleaner than before, allowing it to be extended easily in the future. In this section, we take a look at our implementation from the point of view of an Agda user (Section 3.5.1) and an Agda developer (Section 3.5.2).

3.5.1 Impact on the Agda user

From the point of view of a user of Agda, unification happens behind the scenes while checking definitions by pattern matching, so a different algorithm doesn't impact the syntax of the language directly. Instead, the main criterion a user of Agda should judge the unification algorithm by is that it accepts the definitions that should be accepted, and rejects the definitions that should be rejected. The latter can be seen from the fact that our implementation directly resulted in a fix for issue #1408 (Example 3.33), dealing with an incompatibility between heterogeneous equations and the `--without-K` option (Vezzosi, 2015). Equally important, our implementation provides a much more principled solution to issues #292 (Danielsson, 2010, see also Example 3.33), #1071 (Danielsson, 2014), #1406 (Abel, 2015a, see also Example 3.32), #1411 (Abel, 2015b), and #1427 (Abel, 2015c, see also Example 3.32). All these issues are fixed without introducing special cases in the code and without limiting the power of the unification algorithm in any significant way, as can be seen from the fact that Agda's test suite and standard library are still typechecked correctly. This is in contrast to the previous ad-hoc fixes to some of these issues, which broke the unification algorithm in some cases, for example in issue #1435 (Danielsson, 2015).

The addition of the new unification rules for η-equality of record values also significantly improved the way Agda handles records. Before these unification rule were added to Agda, all variables of record type had to be fully eta-expanded before calling the unifier, for example in issue #473 (Danielsson, 2011). This caused a substantial overhead when dealing with deeply nested records, see issue #635 (Peebles, 2012). This also caused problems in combination with Agda's instance search mechanism, see for example issue #1613 (Abel, 2015d). In contrast, by using this unification rule we only eta-expand a variable when it is useful for the unification to proceed, thus eliminating this overhead.

We also implemented higher-dimensional unification (Section 3.4). This addition allows Agda to typecheck more definitions, such as the example given in issue #1775 (Sicard-Ramírez, 2016).

3.5.2 Impact on the Agda codebase

For the further development of Agda, it is important that the unification machinery is robust and easily extensible with further rules. For this reason, we separated it into two logical parts: a *unification strategy* and the *unification engine*. Both parts make use of the same data structures for representing the unification state and unification rules, as shown in Figure 3.4. The unification strategy takes a unification state as an argument and produces a lazy monadic list of unification rules to try (Figure 3.5), while the unification engine tries to apply these rules one by one until one succeeds (Figure 3.6).

A big difference between our implementation and Agda's previous unification algorithm is that our version explicitly manipulates telescopes of free variables (`varTel`) and equations (`eqTel`) as well as explicit substitutions between these telescopes, while previously these had to be reconstructed after unification was finished. This change resulted in a significant simplification of the code for checking left-hand sides and coverage of definitions by pattern matching (the parts of Agda that use the unification algorithm).

An important choice when constructing a unification strategy is whether to start on the leftmost or the rightmost equation. It seems sensible to start on the left to avoid heterogeneous equations as much as possible, and this was also the preferred method for the old algorithm. However, our unification rules for indexed datatypes actually benefit from having unsolved equations in the telescope, so a unification strategy that starts from the right provides more opportunities to apply these rules. For this reason, our current implementation uses a right-to-left strategy, although plugging in a different strategy would be trivial.

Our implementation of higher-dimensional unification closely follows the steps in Section 3.4.2. In particular, when applying the injectivity rule to a unification problem of the form $(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}_e} \mathtt{c}\ \bar{t}_2)$ the unification algorithm constructs the new unification problem $\Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$ where $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$ and $\bar{v}$ is the lower-dimensional analogue of $\bar{v}_e$, and recursively calls itself on this new problem.

One noteworthy fact about the implementation is how the left- and right-hand sides $f\ \bar{s}_1\ \bar{t}_1\ \overline{\mathtt{refl}}$ and $f\ \bar{s}_2\ \bar{t}_2\ \overline{\mathtt{refl}}$ of the new unification problem in (3.57) are computed. The implementation doesn't have an explicit representation of the

```
data UnifyState = UState
  { varTel   :: Telescope
  , flexVars :: FlexibleVars
  , eqTel    :: Telescope
  , eqLHS    :: [Term]
  , eqRHS    :: [Term]
  }

data UnifyStep
  = Deletion               { ... }
  | Solution               { ... }
  | Injectivity            { ... }
  | Conflict               { ... }
  | Cycle                  { ... }
  | EtaExpandVar           { ... }
  | EtaExpandEquation      { ... }
  | LitConflict            { ... }
  | StripSizeSuc           { ... }
  | SkipIrrelevantEquation { ... }
  | TypeConInjectivity     { ... }
```

Figure 3.4: The datatypes used for representing unification states and unification rules closely follow the theory. In addition to the unification rules presented in this chapter, Agda also has unification rules for dealing with literals, sized types (Abel, 2010) and irrelevant equations (Abel, 2011), features not discussed in this thesis. There is also a rule for injective type constructors that is only used when this is enabled explicitly by the user.

```
type UnifyStrategy =
  UnifyState -> ListT TCM UnifyStep

skipIrrelevantStrategy basicUnifyStrategy
dataStrategy literalStrategy etaExpandVarStrategy
etaExpandEquationStrategy injectiveTypeConStrategy
simplifySizesStrategy checkEqualityStrategy
  :: Int -> UnifyStrategy
```

Figure 3.5: A unification strategy takes a unification state and produces a list of unification steps to try in order. For constructing unification strategies, we provide a number of basic strategies that can be combined in any order.

```
unifyStep :: UnifyState -> UnifyStep
          -> UnifyM (UnificationResult' UnifyState)

unify :: UnifyState -> UnifyStrategy
      -> UnifyM (UnificationResult' UnifyState)
```

Figure 3.6: The unification engine consists of an auxiliary function `unifyStep` that tries to apply one unification step, resulting in either a new state, an absurdity (e.g. for the conflict and cycle rules), or a failure, and the main function `unify` that tries all steps suggested by a given strategy, and continues until either the unification problem is solved (i.e. the equation telescope is empty) or there are no more rules left to try.

function f, so it's not possible to calculate them directly. Instead, the recursive call produces a substitution ρ of type $\Delta' \to \Phi\Delta$. This allows us to calculate $f^{-1} : \Delta' \to (\bar{x} : \Phi)(\bar{y} : \Delta)(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$ as $\lambda\bar{x}'.\ (\bar{x}'\rho; \overline{\texttt{refl}})$, but doesn't give us a direct way to compute f. To go in the opposite direction, we note that ρ is a *pattern* with free variables Δ'. So we can match the values from $\Phi\Delta$ against this pattern. The proofs of $\bar{u} \equiv_\Xi \bar{v}$ (assumed to be $\overline{\texttt{refl}}$ in our implementation, since all the unifiers we compute are strong unifiers) ensure that this matching cannot fail, so this allows us to recover the values of the variables in Δ', thus computing the function $f : \Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v}) \to \Delta'$.

3.6 Related work

Unification is a large area of research that we cannot hope to cover here in full. We refer the interested reader to Jouannaud and Kirchner (1990) and Baader and Snyder (2001) for a general overview of the subject. Most extensions to unification that are studied, such as higher-order unification and E-unification, are orthogonal to the work in this chapter, although it would be interesting to see how they fit within our framework.

Type checkers of dependently typed languages typically have some facility for meta-variables that are solved by higher order pattern unification (Miller, 1991). These kind of unification algorithms differ from the one presented in this chapter because they have to satisfy different requirements. Whereas they search for a unifier that is the most general one among all unifiers that make the two sides *definitionally* equal, our unification algorithm guarantees that it is the most general one among all the ones that make the two sides *propositionally* equal. This affects the kind of unification rules that can be applied. For example, these

unification algorithms suppose all rigid symbols (including type constructors) to be 'injective' for the purpose of unification. Some algorithms even consider defined functions to be rigid (Ziliani and Sozeau, 2015) or make use of user-provided hints to choose one solution over the other (Asperti, Ricciotti, Coen, and Tassi, 2009), thereby giving up on finding most general unifiers entirely in favour of finding solutions more often. In this case, the only problem is that the solution to the metavariable may not be what the user intended. In contrast, our algorithm produces evidence of unification internal to the theory we're working in, and it is actually important that the unifier found by the algorithm is indeed the most general one (otherwise we might lose e.g. coverage of functions by pattern matching). Still, it would be interesting to further investigate the similarities and differences between these two unification algorithms.

Goguen (1989) takes a categorical view on unification, representing most general unifiers as *equalizers* in a category of types and substitutions. It shouldn't be surprising that many of the category-theoretic notions are analogous to the type-theoretic ones presented in this chapter. For example, giving an explicit type to the domain of substitutions helps to avoid problems with non-uniqueness in the definition of a most general unifier in other presentations. Compared to the category-theoretical presentation of unification, our work adds support for indexed datatypes, and it also differs in the fact that type theory allows an internal representation of equations as (telescopic) equality types.

The idea to represent unification problems at the object level by using the identity type stems from McBride (1998b). In McBride's paper, the types of equations are limited to simple (non-dependent) types, and the injectivity rule is likewise limited to simple datatypes. Later, he solves this by introducing a heterogeneous identity type (McBride, 2002). However, the K rule is needed to turn heterogeneous equalities back into homogeneous ones. Additionally, postponing equations is not supported, as heterogeneous equations can only be turned into homogeneous ones if the types are equal. In our previous work, we solved the problem of requiring K, but the unification rules still only worked on the first equation in a telescope (Cockx et al., 2014b). As a consequence, we had to limit the injectivity, conflict, and cycle rules to work only in homogeneous situations, while here we can use them in their fully general form.

Our approach to unification is closely related to the notion of inversion of an inductive hypothesis (Cornes and Terrasse, 1995; Monin, 2010). The usual approach to inversion works by crafting a *diagonalizer* that is used as the motive for an eliminator. Unification can also be as an alternative method for proving inversion lemmas (McBride, 1998b). One advantage of the diagonalizer approach is that it moves most of the work to the type level, potentially improving performance of the resulting function. The process of constructing diagonalizers has recently also been automated (Braibant, 2013). However, it

requires that the indices of the inductive hypothesis we are inverting can be written as a pattern, which is not always the case (e.g. they may be non-linear), so the approach based on unification seems to be more general. It would be interesting to try to implement an inversion tactic based on the unification algorithm in this chapter to compare the power of the two approaches.

The idea to view equality proofs themselves as the subjects of unification is inspired by cubical type theory, where equality proofs are terms viewed 'one level up' (Cohen et al., 2016). In fact, if we were working in a cubical type theory, there would be no difference between regular unification and higher-dimensional unification, so the work in this chapter could be seen as 'backporting' some of the power of cubical type theory back to the (currently) better-understood world of standard intuitionistic type theory.

Compared to our reverse unification rules from Cockx et al. (2016a), higher-dimensional unification takes information into account from the types of the constructors as well as the types of the equation. This difference is similar to the inversion of an inductive hypothesis by using a diagonalizer (Cornes and Terrasse, 1995) versus using unification for the problem (McBride, 1998b).

Chapter 4

Back to eliminators

> *Perfection is achieved not when there is nothing left to add, but when there is nothing left to take away.*
>
> — Antoine de Saint-Exupéry (1939)

In this chapter, we take everything we have done in the previous chapters and explain it in terms of the 'bare metal' of type theory: datatype eliminators. These eliminators encode the basic induction principles associated to each datatype. By translating definitions by pattern matching to eliminators, we can be confident that they don't add anything extra to the core theory besides a more convenient syntax (Theorem 4.27).

When translating one concept to another, there is always the danger that some of the original meaning gets lost in the translation. For example, we may accidentally translate a function `not` : `Bool` → `Bool` to a function `not'` : `Bool` → `Bool` of the same type but for which `not' true` = `true`. More subtly, the translated definition could compute the same results for closed terms but have a different computational behaviour when applied to open terms (i.e. ones with free variables). For example, a function `f` : `Bool` → `Bool` may satisfy $f\ x = x$ while the translated function `f'` only satisfies `f' true` = `true` and `f' false` = `false` but not `f'` $x = x$ for arbitrary x. This becomes important when we want to prove properties of this function: instead of writing `refl` for the proof of `f'` $x \equiv_{\texttt{Bool}} x$, we would instead have to handle the two cases `f' true` $\equiv_{\texttt{Bool}}$ `true` and `f' false` $\equiv_{\texttt{Bool}}$ `false` separately. To avoid mismatches between the definition by pattern matching and its translated version, we prove that the translated version preserves the same definitional equalities as the original (Theorem 4.29).

Our proof mostly follows the translation from pattern matching to eliminators by Goguen et al. (2006). The general idea of the proof is as follows. First, the definition by pattern matching is translated to a case tree as explained in Section 2.2.1, using the unification algorithm presented in Chapter 3. Each leaf node of the case tree corresponds to a clause $\mathtt{f}\ \bar{p} = e$, i.e. it defines $\mathtt{f}$ on arguments that match the pattern $\bar{p}$, and each internal node corresponds to a case split of $\bar{p}$ on some variable $x : \mathtt{D}\ \bar{u}$ into patterns $\bar{p}_1, \ldots, \bar{p}_n$. If we can assemble the definitions of $\mathtt{f}\ \bar{p}_1, \ldots, f\ \bar{p}_n$ into a definition of $\mathtt{f}\ \bar{p}$, then we can work backwards from the leaf nodes towards the root, ultimately obtaining a definition of $\mathtt{f}$ on arbitrary variables.

To assemble the definitions of $\mathtt{f}\ \bar{p}_1, \ldots, \mathtt{f}\ \bar{p}_n$ into a definition of $\mathtt{f}\ \bar{p}$, we proceed in two steps. First we apply a technique called *basic* $\mathtt{case}_{\mathtt{D}}$*-analysis*. This splits the problem into one subproblem for each constructor c_i of $\mathtt{D}$, and generates new equations between the indices of the datatype. The second step is to apply *specialization by unification*, simplifying these equations step by step. The unification transitions make sure that we do not have to fill in anything for a negative success. Finally, we fill in the translated definition of $\mathtt{f}\ \bar{p}_i$ for each positive success.

In general there can be recursive calls to the function $\mathtt{f}$ in each clause $\mathtt{f}\ \bar{p} = e$. These recursive calls are required to be structurally recursive on some argument $x : \mathtt{D}\ \bar{u}$ of $\mathtt{f}$. This allows us to use well-founded recursion on $\mathtt{D}$ to obtain an inductive hypothesis H, asserting that $\mathtt{f}$ is already defined on arguments structurally smaller than x. This inductive hypothesis is then used to replace the recursive calls to $\mathtt{f}$ in e.

The challenge is then to construct all these techniques as terms *internal to type theory*. We start by constructing each of these techniques in turn: case splitting (Section 4.1.1), structural recursion (Section 4.1.2), and the injectivity, conflict, and cycle rules from Chapter 3 (Section 4.1.3 and Section 4.1.4). These constructions are based on those from McBride et al. (2006), but they are adapted to use our definition of telescopic equality in order to take the additional dependencies on equality proofs into account. Then we present basic analysis (Section 4.2.1) and specialization by unification (Section 4.2.2). Finally, all these tools are brought together for the translation of case trees to eliminators (Section 4.3).

4.1 Basic constructions on constructors

As the target theory of our translation, we take the same basic theory as given in Section 2.1, except that in this theory it is not allowed to define functions by

pattern matching. Instead, the only way to define a function by recursion or induction is by using the appropriate *datatype eliminator*.

Datatype eliminators are an instance of the more general concept of an *elimination operator* (McBride, 2002).

Definition 4.1 (Elimination operator)**.** Let Ψ be any telescope. A Ψ-*elimination operator* is any function with a type of the form

$$\begin{aligned}&(P : \Psi \to \mathtt{Set}_i) \to \\ &(m_1 : \Delta_1 \to P\ \bar{s}_1) \ldots (m_n : \Delta_n \to P\ \bar{s}_n) \to \\ &(\bar{t} : \Psi) \to P\ \bar{t}\end{aligned} \tag{4.1}$$

We call Ψ the *target*, P the *motive*, and $m_1, \ldots, m_n$ the *methods* of the elimination operator.

We can think of a Ψ-elimination operator as a way to transform a problem into a set of subproblems. In the type shown above, the problem is to construct a result of type $P\ \bar{t}$ when given arbitrary values $\bar{t}$ in the telescope Ψ. This original problem is transformed into n sub-problems given by each of the methods: the ith subproblem is to construct a result of type $P\ \bar{s}_i$ when given arbitrary values of type Δ_i. The elimination operator's type can be read as a function that transforms solutions for the sub-problems into a solution for the original problem.

For the rest of this section, let $\mathtt{D} : \Xi \to \mathtt{Set}_i$ be an inductive family (where Ξ is the telescope of the indices) with constructors $\mathtt{c}_1, \ldots, \mathtt{c}_k$. Without loss of generality, we assume that the non-recursive constructor arguments come before the recursive ones, so $\mathtt{c}_i$ has type:

$$\mathtt{c}_i : \Delta_i \to (\Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1}) \to \ldots \to (\Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \mathtt{D}\ \bar{u}_i \tag{4.2}$$

As before, we consider $\mathtt{D}$ to be already applied to its parameters, if it has any.

Definition 4.2 (Datatype eliminator)**.** The standard datatype eliminator $\mathtt{elim}_\mathtt{D}$ for $\mathtt{D}$ is a $\overline{\mathtt{D}}$-eliminator of type

$$\begin{aligned}\mathtt{elim}_\mathtt{D}\ :\ &(P : \overline{\mathtt{D}} \to \mathtt{Set}_i)(m_1 : M_1) \ldots (m_k : M_k) \\ \to\ &(\bar{x} : \overline{\mathtt{D}}) \to P\ \bar{x}\end{aligned} \tag{4.3}$$

where the methods $m_1, \ldots, m_k$ have type

$$\begin{aligned}M_i\ =\ &(\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \\ &\to (h_1 : (\bar{s}_1 : \Phi_{i,1}) \to P\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \to \ldots \\ &\to (h_{n_i} : (\bar{s}_{n_i} : \Phi_{i,n_i}) \to P\ \bar{v}_{i,n_i}\ (x_{n_i}\ \bar{s}_{n_i})) \\ &\to P\ \bar{u}_i\ (\mathtt{c}_i\ \bar{t}\ x_1\ \ldots\ x_{n_i})\end{aligned} \tag{4.4}$$

The evaluation behaviour of the standard datatype eliminator is given by the following rule for $i = 1, \ldots, k$:

$$\begin{aligned}&\mathtt{elim_D}\ P\ m_1\ \ldots\ m_k\ \bar{u}_i\ (\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) = \\ &\quad m_i\ \ \bar{t}\ x_1\ \ldots\ x_{n_i} \\ &\qquad (\lambda \bar{s}_1.\, \mathtt{elim_D}\ P\ m_1\ \ldots\ m_k\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \\ &\qquad \ldots \\ &\qquad (\lambda \bar{s}_{n_i}.\, \mathtt{elim_D}\ P\ m_1\ \ldots\ m_k\ \bar{v}_{i,n_i}(x_{n_i}\ \bar{s}_{n_i}))\end{aligned} \tag{4.5}$$

Example 4.3. Consider the datatype `Tree` of binary trees:

$$\begin{aligned}&\textbf{data}\ \mathtt{Tree} : \mathtt{Set}\ \textbf{where} \\ &\quad \mathtt{leaf}\ :\ \mathtt{Tree} \\ &\quad \mathtt{node}\ :\ \mathtt{Tree} \to \mathtt{Tree} \to \mathtt{Tree}\end{aligned} \tag{4.6}$$

The eliminator for `Tree` is

$$\begin{aligned}\mathtt{elim_{Tree}}\ :\ &(P : \mathtt{Tree} \to \mathtt{Set}_i)(m_{\mathtt{leaf}} : P\ \mathtt{leaf}) \\ \to\ &(m_{\mathtt{node}} : (l\ r : \mathtt{Tree}) \to P\ l \to P\ r \to P\ (\mathtt{node}\ l\ r)) \\ \to\ &(x : \mathtt{Tree}) \to P\ x\end{aligned} \tag{4.7}$$

The evaluation rules are

$$\mathtt{elim_{Tree}}\ P\ m_{\mathtt{leaf}}\ m_{\mathtt{node}}\ \mathtt{leaf} = m_{\mathtt{leaf}} \tag{4.8}$$

and

$$\begin{aligned}&\mathtt{elim_{Tree}}\ P\ m_{\mathtt{leaf}}\ m_{\mathtt{node}}\ (\mathtt{node}\ l\ r) = \\ &\quad m_{\mathtt{node}}\ l\ r\ (\mathtt{elim_{Tree}}\ P\ m_{\mathtt{leaf}}\ m_{\mathtt{node}}\ l)\ (\mathtt{elim_{Tree}}\ P\ m_{\mathtt{leaf}}\ m_{\mathtt{node}}\ r)\end{aligned} \tag{4.9}$$

Example 4.4. Consider the indexed datatype $m \leq n$ (Example 2.12):

$$\begin{aligned}&\textbf{data}\ \_ \leq \_ : \mathbb{N} \to \mathbb{N} \to \mathtt{Set}\ \textbf{where} \\ &\quad \mathtt{lz}\ :\ (n : \mathbb{N}) \to \mathtt{zero} \leq n \\ &\quad \mathtt{ls}\ :\ (m\ n : \mathbb{N}) \to m \leq n \to \mathtt{suc}\ m \leq \mathtt{suc}\ n\end{aligned} \tag{4.10}$$

The eliminator for $\leq$ is

$$\begin{aligned}\mathtt{elim}_{\leq}\ :\ &(P : (m : \mathbb{N})(n : \mathbb{N})(x : m \leq n) \to \mathtt{Set}_i) \\ \to\ &(m_{\mathtt{lz}} : (n : \mathbb{N}) \to P\ \mathtt{zero}\ n\ (\mathtt{lz}\ n)) \\ \to\ &(m_{\mathtt{ls}} : (m : \mathbb{N})(n : \mathbb{N})(x : m \leq n) \to P\ m\ n\ x \\ &\qquad \to P\ (\mathtt{suc}\ m)\ (\mathtt{suc}\ n)\ (\mathtt{ls}\ m\ n\ x)) \\ \to\ &(m : \mathbb{N})(n : \mathbb{N})(x : m \leq n) \to P\ m\ n\ x\end{aligned} \tag{4.11}$$

The evaluation rules are

$$\mathtt{elim}_{\leq}\ P\ m_{\mathtt{lz}}\ m_{\mathtt{ls}}\ \mathtt{zero}\ n\ (\mathtt{lz}\ n) = m_{\mathtt{lz}}\ n \tag{4.12}$$

and

$$\begin{aligned}&\mathtt{elim}_{\leq}\ P\ m_{\mathtt{lz}}\ m_{\mathtt{ls}}\ (\mathtt{suc}\ m)\ (\mathtt{suc}\ n)\ (\mathtt{ls}\ m\ n\ x) = \\ &\quad m_{\mathtt{ls}}\ m\ n\ x\ (\mathtt{elim}_{\leq}\ P\ m_{\mathtt{lz}}\ m_{\mathtt{ls}}\ m\ n\ x)\end{aligned} \tag{4.13}$$

4.1.1 Case analysis

The most important part of dependent pattern matching is that it allows us to do case analysis on a variable of an inductive type, returning a different value for each constructor. In particular, each internal node of a case tree corresponds exactly to one case split. In the translated version of the definition, each case split corresponds to an application of the $\mathtt{case}_{\mathtt{D}}$-eliminator. In effect, it is a weaker version of the standard eliminator, with the inductive hypotheses for the recursive arguments dropped.

Lemma 4.5 ($\mathtt{case}_{\mathtt{D}}$)**.** *We have a function* $\mathtt{case}_{\mathtt{D}}$ *of type*

$$\begin{aligned}\mathtt{case}_{\mathtt{D}} \;:\; & (P : \overline{\mathtt{D}} \to \mathtt{Set}_i)(m_1 : M_1) \ldots (m_k : M_k) \\ \to\; & (\bar{x} : \overline{\mathtt{D}}) \to P\ \bar{x}\end{aligned} \tag{4.14}$$

where

$$\begin{aligned}M_i \;:\; & (\bar{t} : \Delta_i) \to (x_1 : \Phi_{i,1} \to D\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to D\ \bar{v}_{i,n_i}) \\ & \to P\ \bar{u}_i\ (\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i})\end{aligned} \tag{4.15}$$

for $i = 1, \ldots k$.

Example 4.6. For the `Tree` type, we have:

$$\begin{aligned}&\mathtt{case}_{\mathtt{Tree}} \;:\; (P : \mathtt{Tree} \to \mathtt{Set}_i) \to P\ \mathtt{leaf} \\ &\qquad\qquad \to\; ((l\ r : \mathtt{Tree}) \to P\ (\mathtt{node}\ l\ r)) \to (x : \mathtt{Tree}) \to P\ x \\ &\mathtt{case}_{\mathtt{Tree}}\ P\ m_{\mathtt{leaf}}\ m_{\mathtt{node}}\ t = \mathtt{elim}_{\mathtt{Tree}}\ P\ m_{\mathtt{leaf}}\ (\lambda l\ r\ h_l\ h_r.\ m_{\mathtt{node}}\ l\ r)\ t\end{aligned} \tag{4.16}$$

Example 4.7. For the type $m \leq n$, we have:

$$\begin{aligned}\mathtt{case}_{\leq} \;:\; & (P : (m : \mathbb{N})(n : \mathbb{N})(x : m \leq n) \to \mathtt{Set}_i) \\ \to\; & (m_{\mathtt{lz}} : (n : \mathbb{N}) \to P\ \mathtt{zero}\ n\ (\mathtt{lz}\ n)) \\ \to\; & (m_{\mathtt{ls}} : (m : \mathbb{N})(n : \mathbb{N})(x : m \leq n) \to P\ (\mathtt{suc}\ m)\ (\mathtt{suc}\ n)\ (\mathtt{ls}\ m\ n\ x)) \\ \to\; & (m : \mathbb{N})(n : \mathbb{N})(x : m \leq n) \to P\ m\ n\ x\end{aligned} \tag{4.17}$$

Construction of $\mathtt{case}_{\mathtt{D}}$.

$$\mathtt{case}_{\mathtt{D}}\ P\ m_1\ \ldots\ m_k = \mathtt{elim}_{\mathtt{D}}\ P\ (\lambda \bar{t}\ \bar{x}\ \bar{h}.\, m_1\ \bar{t}\ \bar{x})\ \ldots\ (\lambda \bar{t}\ \bar{x}\ \bar{h}.\, m_k\ \bar{t}\ \bar{x}) \tag{4.18}$$

□

4.1.2 Structural recursion

A second core feature of dependent pattern matching is that it allows us to write definitions by well-founded recursion: a function definition can make recursive calls to itself applied not just to immediate subterms but also sub-subterms, sub-sub-subterms, etcetera. For example, the definition $\mathtt{f}\ (\mathtt{suc}\ (\mathtt{suc}\ (\mathtt{suc}\ n)))$ can make use of $(\mathtt{f}\ (\mathtt{suc}\ (\mathtt{suc}\ n)))$, $(\mathtt{f}\ (\mathtt{suc}\ n))$ and $(\mathtt{f}\ n)$. In the translated version, structural recursion is translated to an application of the $\mathtt{rec}_{\mathtt{D}}$-eliminator.

Before we define $\mathtt{rec}_{\mathtt{D}}$, we first define the auxiliary type $\mathtt{Below}_{\mathtt{D}}$ collecting all possible recursive calls: $\mathtt{Below}_{\mathtt{D}}\ P\ \bar{u}\ x$ is defined as a tuple type that is inhabited whenever $P\ \bar{v}\ y$ holds for all $y : \mathtt{D}\ \bar{v}$ that are structurally smaller than $x : \mathtt{D}\ \bar{u}$

Lemma 4.8 ($\mathtt{Below}_{\mathtt{D}}$)**.** *Let $P : \overline{\mathtt{D}} \to \mathtt{Set}_i$. For any $x : \mathtt{D}\ \bar{u}$, we have a type $\mathtt{Below}_{\mathtt{D}}\ P\ \bar{u}\ x$ such that for any $y \prec x$ we have a projection $\pi : \mathtt{Below}_{\mathtt{D}}\ P\ \bar{u}\ x \to P\ \bar{v}\ y$.*

Example 4.9. The type $\mathtt{Below}_{\mathtt{Tree}}\ P\ x$ expresses that the property $P : \mathtt{Tree} \to \mathtt{Set}$ holds for any subtree of $x : \mathtt{Tree}$. In other words, we have

$$\begin{aligned} \mathtt{Below}_{\mathtt{Tree}}\ P\ \mathtt{leaf} &= \top \\ \mathtt{Below}_{\mathtt{Tree}}\ P\ (\mathtt{node}\ l\ r) &= (\mathtt{Below}_{\mathtt{Tree}}\ P\ l \times P\ l) \times (\mathtt{Below}_{\mathtt{Tree}}\ P\ r \times P\ r) \end{aligned} \tag{4.19}$$

Construction of $\mathtt{Below}_{\mathtt{D}}\ P$. We apply the eliminator $\mathtt{elim}_{\mathtt{D}}$ to the motive $\Phi = \lambda\ \_.\ \mathtt{Set}_i$. For the method m_i corresponding to the constructor $\mathtt{c_i}$ we give the following:

$$\begin{aligned} m_i \;=\; & \lambda \bar{t}; x_1; \ldots; x_{n_i}; h_1; \ldots; h_{n_i}. \\ & ((\bar{s}_1 : \Phi_{i,1}) \to h_1\ \bar{s}_1 \times P\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \times \\ & \cdots \times ((\bar{s}_{n_i} : \Phi_{i,n_i}) \to h_{n_i}\ \bar{s}_{n_i} \times P\ \bar{v}_{i,n_i}\ (x_{n_i}\ \bar{s}_{n_i})) \end{aligned} \tag{4.20}$$

To construct the projection π, consider $x : \mathtt{D}\ \bar{u}$ and any structurally smaller term $y : \mathtt{D}\ \bar{v}$. If y is (an application of) a direct subterm of x, say $x = \mathtt{c}\ \bar{t}\ x_1\ \ldots\ x_n$ with $y = x_i\ \bar{w}$, then we return the second component of the ith component of $\mathtt{Below}_{\mathtt{D}}\ P\ x$, i.e. we define

$$\pi\ H = \pi_2\ (\pi_i\ H\ \bar{w}) : \mathtt{Below}_{\mathtt{D}}\ P\ \bar{u}\ x \to P\ \bar{v}\ y \tag{4.21}$$

Otherwise, y is a subterm of some direct subterm x_i of $x = \mathtt{c}\ \bar{t}\ x_1\ \ldots\ x_n$. In particular, by induction we have some $\pi' : \mathtt{Below}_{\mathtt{D}}\ P\ \bar{v}_i\ x_i \to P\ \bar{v}\ y$. This allows us to define π as follows:

$$\pi\ H = \pi'\ (\pi_1\ (\pi_i\ H)) : \mathtt{Below}_{\mathtt{D}}\ P\ \bar{u}\ x \to P\ \bar{v}\ y \tag{4.22}$$

□

It is possible to define an internal version of $x \prec t$ as a type in a similar way to $\mathtt{Below_D}$, replacing the product types $\times$ by sum types $\uplus$. However, this is not necessary as this type never shows up in the translated version of a definition by pattern matching, it is only used to specify when the translation can be performed.

Lemma 4.10 ($\mathtt{below_D}$)**.** *We have a function* $\mathtt{below_D}$ *of type*

$$\begin{aligned}\mathtt{below_D} \;:\; & (P : (\bar{x} : \overline{\mathtt{D}}) \to \mathtt{Set}_i)(p : (\bar{x} : \overline{\mathtt{D}}) \to \mathtt{Below_D}\ P\ \bar{x} \to P\ \bar{x}) \\ \to\; & (\bar{x} : \overline{\mathtt{D}}) \to \mathtt{Below_D}\ P\ \bar{x}\end{aligned} \tag{4.23}$$

Definition 4.11 ($\mathtt{rec_D}$)**.** We define $\mathtt{rec_D}$ of type

$$(P : (\bar{x} : \overline{\mathtt{D}}) \to \mathtt{Set}_i)(p : (\bar{x} : \overline{\mathtt{D}}) \to \mathtt{Below_D}\ P\ \bar{x} \to P\ \bar{x})(\bar{x} : \overline{\mathtt{D}}) \to P\ \bar{x} \tag{4.24}$$

by $\mathtt{rec_D}\ P\ p\ \bar{x} := p\ \bar{x}\ (\mathtt{below_D}\ P\ p\ \bar{x})$.

Construction of $\mathtt{below_D}$. To construct $\mathtt{below_D}\ P\ p$, we apply $\mathtt{elim_D}$ with the motive $\mathtt{Below_D}\ P$. The method m_i is required to have type

$$\begin{aligned} m_i \;:\; & (\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \\ \to\; & (h_1 : (\bar{s}_1 : \Phi_{i,1}) \to \mathtt{Below_D}\ P\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \to \ldots \\ \to\; & (h_{n_i} : (\bar{s}_{n_i} : \Phi_{i,n_i}) \to \mathtt{Below_D}\ P\ \bar{v}_{i,n_i}\ (x_{n_i}\ \bar{s}_{n_i})) \\ \to\; & \mathtt{Below_D}\ P\ \bar{u}_i\ (\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \end{aligned} \tag{4.25}$$

that we construct as follows:

$$\begin{aligned} m_i\ \bar{t}; x_1; \ldots; x_{n_i}; h_1; \ldots; h_{n_i} \;=\; & \lambda \bar{s}_1.\ (h_1\ \bar{s}_1,\ p\ \bar{v}_{i,1}\ x_1\ (h_1\ \bar{s}_1)), \ldots, \\ & \lambda \bar{s}_{n_i}.\ (h_{n_i}\ \bar{s}_{n_i},\ p\ \bar{v}_{i,n_i}\ x_{n_i}\ (h_{n_i}\ \bar{s}_{n_i})) \end{aligned} \tag{4.26}$$

□

Example 4.12. The function $\mathtt{below_{Tree}}$ encodes proof by complete induction on trees: if we can give a step function s that proves $\mathtt{Below_{Tree}}\ P\ t$ implies $P\ t$ for any tree t, then $\mathtt{below_{Tree}}\ P\ s$ is a proof that $\mathtt{Below_{Tree}}\ P\ t$ holds for any t. The recursion operator $\mathtt{rec_{Tree}}\ P\ s$ applies the step function s one more time to conclude $P\ t$ for any tree t.

Lemma 4.13. *Let* $x : \mathtt{D}\ \bar{u}$ *and* $y : \mathtt{D}\ \bar{v}$ *such that* $y \prec x$ *and let* $\pi : \mathtt{Below_D}\ P\ \bar{u}\ x \to P\ \bar{v}\ y$ *be the projection given by Lemma 4.8. For any* $P : (\bar{x} : \overline{\mathtt{D}}) \to \mathtt{Set}_i$ *and* $p : (\bar{x} : \overline{\mathtt{D}}) \to \mathtt{Below_D}\ P\ \bar{x} \to P\ \bar{x}$*, we have the definitional equality:*

$$\pi\ (\mathtt{below_D}\ P\ p\ \bar{u}\ x) = \mathtt{rec_D}\ P\ p\ \bar{v}\ y \tag{4.27}$$

Proof. We prove this by induction on the derivation of $y \prec x$. There are two cases: either y is (an application of) a direct subterm of x, or y is a subterm of a direct subterm of x. In both cases, x is of the form $\mathtt{c}\ \bar{t}\ x_1\ \ldots\ x_n : \mathtt{D}\ \bar{u}$, so by definition of $\mathtt{below}_\mathtt{D}$ we have

$$\begin{aligned}&\mathtt{below}_\mathtt{D}\ P\ p\ \bar{u}\ (\mathtt{c}\ \bar{t}\ x_1\ \ldots\ x_n)\\ &\quad = \lambda \bar{s}_1.\ (\mathtt{below}_\mathtt{D}\ P\ p\ \bar{v}_1\ (x_1\ \bar{s}_1),\ p\ \bar{v}_1\ x_1\ (\mathtt{below}_\mathtt{D}\ P\ p\ \bar{v}_1\ (x_1\ \bar{s}_1))), \ldots,\\ &\qquad \lambda \bar{s}_n.\ (\mathtt{below}_\mathtt{D}\ P\ p\ \bar{v}_n\ (x_n\ \bar{s}_n),\ p\ \bar{v}_n\ x_n\ (\mathtt{below}_\mathtt{D}\ P\ p\ \bar{v}_n\ (x_n\ \bar{s}_n)))\end{aligned} \tag{4.28}$$

In the first case y is (an application of) a direct subterm of x, i.e. $y = x_i\ \bar{w}$, so $\pi\ H = \pi_2\ (\pi_i\ H\ \bar{w}) = p\ \bar{v}_i\ x_i\ (\mathtt{below}_\mathtt{D}\ P\ p\ \bar{v}\ (x_i\ \bar{w})) = \mathtt{rec}_\mathtt{D}\ P\ p\ \bar{v}\ (x_i\ \bar{w}) = \mathtt{rec}_\mathtt{D}\ P\ p\ \bar{v}\ y$.

In the second case, we have $y \prec x_i$, so $\pi\ H = \pi'\ (\pi_1\ (\pi_i\ H)) = \pi'\ (\mathtt{below}_\mathtt{D}\ P\ p\ \bar{v}_i\ x_i)$ where $\pi' : \mathtt{Below}_\mathtt{D}\ P\ \bar{v}_i\ x_i \to P\ \bar{v}\ y$. By induction, we have $\pi'\ (\mathtt{below}_\mathtt{D}\ P\ p\ \bar{v}_i\ x_i) = \mathtt{rec}_\mathtt{D}\ P\ p\ \bar{v}\ y$. □

4.1.3 No confusion

To perform case analysis on a variable whose indices are not fully general, we apply unification. Two of the unification rules, injectivity and conflict, are instances of a more general principle known as 'no confusion'. In this section, we construct this principle internally as an equivalence $\mathtt{noConf}_\mathtt{D}$.

As for structural recursion, we first define an auxiliary type in order to give a general type to $\mathtt{noConf}_\mathtt{D}$.

Lemma 4.14 ($\mathtt{NoConfusion}_\mathtt{D}$)**.** *We have a type* $\mathtt{NoConfusion}_\mathtt{D} : \overline{\mathtt{D}} \to \overline{\mathtt{D}} \to \mathtt{Set}_d$ *such that*

$$\begin{aligned}&\mathtt{NoConfusion}_\mathtt{D}\ (\bar{u}; \mathtt{c_i}\ \bar{s})\ (\bar{v}; \mathtt{c_i}\ \bar{t}) &&= \bar{s} \equiv_{\Delta_i} \bar{t}\\ &\mathtt{NoConfusion}_\mathtt{D}\ (\bar{u}; \mathtt{c_i}\ \bar{s})\ (\bar{v}; \mathtt{c_j}\ \bar{t}) &&= \bot \qquad (\textit{when } i \neq j)\end{aligned} \tag{4.29}$$

On the diagonal (where we have two times the same constructor), $\mathtt{NoConfusion}_\mathtt{D}$ only requires $\bar{s} \equiv_{\Delta_\mathtt{c}} \bar{t}$. From this it follows that $\bar{u} \equiv_\Xi \bar{v}$ as well, since the indices are determined by the choice of constructor and its arguments.

Example 4.15. For the `Tree` datatype, $\mathtt{NoConfusion}\ t_1\ t_2$ is defined as follows:

$$\begin{aligned}&\mathtt{NoConfusion} : \mathtt{Tree} \to \mathtt{Tree} \to \mathtt{Set}\\ &\mathtt{NoConfusion}\ \mathtt{leaf}\ \ \mathtt{leaf}\ \ &&= \top\\ &\mathtt{NoConfusion}\ \mathtt{leaf}\ \ (\mathtt{node}\ l\ r) &&= \bot\\ &\mathtt{NoConfusion}\ (\mathtt{node}\ l\ r)\ \ \mathtt{leaf}\ \ &&= \bot\\ &\mathtt{NoConfusion}\ (\mathtt{node}\ l_1\ r_1)\ (\mathtt{node}\ l_2\ r_2) &&= (l_1 \equiv_\mathtt{Tree} l_2) \times (r_1 \equiv_\mathtt{Tree} r_2)\end{aligned} \tag{4.30}$$

Construction of $\mathtt{NoConfusion_D}$. We apply $\mathtt{case_D}$ with the motive $\lambda\ \_.\ \overline{\mathtt{D}} \to \mathtt{Set}_i$. For each method $m_i\ \bar{x}$, we apply $\mathtt{case_D}$ again with motive $\lambda\_ \to \mathtt{Set}$. This gives us k^2 methods $m_{i,j}$ to fill in, one for each pair of constructors. On the diagonal (where $i = j$) we define $m_{ii} = \lambda\bar{x}; \bar{x}'.\ \bar{x} \equiv_{\Delta_i} \bar{x}'$, and if $i \neq j$ we give $m_{i,j} = \lambda\bar{x}; \bar{x}'.\ \bot$. □

Lemma 4.16 ($\mathtt{noConf_D}$)**.** *We have an equivalence*

$$\mathtt{noConf_D} : (\bar{x}\ \bar{y} : \overline{\mathtt{D}}) \to (\bar{x} \equiv_{\overline{\mathtt{D}}} \bar{y}) \simeq \mathtt{NoConfusion_D}\ \bar{x}\ \bar{y} \tag{4.31}$$

Moreover, for any constructor $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$ *and* $\bar{s}, \bar{s}' : \Delta$*, this equivalence satisfies* $\mathtt{noConf_D}^{-1}\ (\bar{u}[\Delta \mapsto \bar{s}]; \mathtt{c}\ \bar{s})\ (\bar{u}[\Delta \mapsto \bar{s}']; \mathtt{c}\ \bar{s}') = \overline{\mathtt{dcong}}\ (\lambda\bar{x}.\ (\bar{u}; \mathtt{c}\ \bar{x}))$.

Example 4.17. For $\mathtt{Tree}$, the function $\mathtt{noConf_{Tree}}$ gives for any two trees s and t that are equal a proof of $\mathtt{NoConfusion_{Tree}}\ s\ t$:

$$\mathtt{noConf_{Tree}} : (s\ t : \mathtt{Tree}) \to (s \equiv_{\mathtt{Tree}} t) \simeq \mathtt{NoConfusion_{Tree}}\ s\ t \tag{4.32}$$

If s and t are of the form $\mathtt{node}\ l_1\ r_1$ and $\mathtt{node}\ l_2\ r_2$ respectively, then this gives us the injectivity rule $(\mathtt{node}\ l_1\ r_1 \equiv_{\mathtt{Tree}} \mathtt{node}\ l_2\ r_2) \simeq (l_1 \equiv_{\mathtt{Tree}} l_2 \times r_1 \equiv_{\mathtt{Tree}} r_2)$. On the other hand, if s is of the form $\mathtt{leaf}$ and t is of the form $\mathtt{node}\ l\ r$, then we get the conflict rule $(\mathtt{leaf} \equiv_{\mathtt{Tree}} \mathtt{node}\ l\ r) \simeq \bot$.

Construction of $\mathtt{noConf_D}$. First, we define the left-to-right function $\mathtt{noConf_D}\ \bar{a}\ \bar{b}$. To do this, we apply telescopic substitution $\overline{\mathtt{subst}}$ with motive $\mathtt{NoConfusion_D}\ \bar{a}$. This reduces the problem to finding a function of type

$$(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{NoConfusion_D}\ \bar{a}\ \bar{a} \tag{4.33}$$

But this can be done using $\mathtt{case_D}$ with motive $\lambda\ \bar{a}.\ \mathtt{NoConfusion_D}\ \bar{a}\ \bar{a}$, filling in $\overline{\mathtt{refl}}$ for each method $m_i\ \bar{x}$.

For the inverse $\mathtt{noConf_D}^{-1}\ \bar{a}\ \bar{b}$, we need to do a little more work. First, we apply $\mathtt{case_D}$ twice as in the definition of $\mathtt{NoConfusion_D}$. Now we are left to give methods

$$m_{i,j} : \mathtt{NoConfusion_D}\ (\bar{u}_i; \mathtt{c_i}\ \bar{x})\ (\bar{u}'_j; \mathtt{c_j}\ \bar{x}') \to \bar{u}_i\ (\mathtt{c_i}\ \bar{x}) \equiv_{\overline{\mathtt{D}}} \bar{u}'_j\ (\mathtt{c_j}\ \bar{x}') \tag{4.34}$$

When $i \neq j$, this is easy: we get an element of type $\bot$ from $\mathtt{NoConfusion_D}$, from which we can conclude anything. On the diagonal (where $i = j$) we get a proof of $\bar{x} \equiv_{\Delta_i} \bar{x}'$. Applying $\overline{\mathtt{dcong}}$ to this equality gives us $(\bar{u}_i; \mathtt{c_i}\ \bar{x}) \equiv_{\overline{\mathtt{D}}} (\bar{u}'_i; \mathtt{c_i}\ \bar{x}')$, which is what we need.

Next, we prove that this is a left inverse by constructing a function of type

$$(\bar{a}\ \bar{b} : \overline{\mathtt{D}})(\bar{e} : \bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b}) \to \mathtt{noConf_D}^{-1}\ \bar{a}\ \bar{b}\ (\mathtt{noConf_D}\ \bar{a}\ \bar{b}\ \bar{e}) \equiv_{\bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b}} \bar{e} \tag{4.35}$$

By $\overline{\mathtt{J}}$, it is sufficient to give a function of type

$$(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{noConf}_{\mathtt{D}}{}^{-1}\ \bar{a}\ \bar{a}\ (\mathtt{noConf}_{\mathtt{D}}\ \bar{a}\ \bar{a}\ \overline{\mathtt{refl}}) \equiv_{\bar{a} \equiv_{\overline{\mathtt{D}}} \bar{a}} \overline{\mathtt{refl}} \tag{4.36}$$

But this we can do by applying $\mathtt{case}_{\mathtt{D}}$ with methods $m_i\ \bar{x} = \overline{\mathtt{refl}}$.

All that's left to do is to prove that it is a right inverse as well. To construct the proof `isRinv` that

$$(\bar{a}\ \bar{b} : \overline{\mathtt{D}})(e : \mathtt{NoConfusion}_{\mathtt{D}}\ \bar{a}\ \bar{b}) \to \mathtt{noConf}\ \bar{a}\ \bar{b}\ (\mathtt{noConf}^{-1}\ \bar{a}\ \bar{b}\ e) \equiv_{\mathtt{NoConfusion}_{\mathtt{D}}\ \bar{a}\ \bar{b}\ e} \tag{4.37}$$

we first apply case analysis on $\bar{a}$ and $\bar{b}$. In the cases where we have two distinct constructors $\mathtt{c_i}$ and $\mathtt{c_k}$, we have $e : \bot$ so we can conclude by $\mathtt{elim}_{\bot}$. In the diagonal cases we have $e : \bar{s} \equiv_{\Delta_i} \bar{t}$. Eliminating these equations with J leaves us with the goal $\overline{\mathtt{refl}} \equiv_{\bar{s} \equiv_{\Delta_i} \bar{s}} \overline{\mathtt{refl}}$, which we solve by giving $\overline{\mathtt{refl}}$. □

4.1.4 Acyclicity

The final property of datatypes we need is acyclicity: a term can never be structurally smaller than itself. This property is used for implementing the cycle detection rule of the unification algorithm. Internally, it is represented by the term $\mathtt{noCycle}_{\mathtt{D}}$. To express its type, we first define what it means for a term to (not) be structurally smaller than some other term.

Lemma 4.18 ($\not<_{\mathtt{D}}$)**.** *We have a type* $\_ \not<_{\mathtt{D}} \_ : \overline{\mathtt{D}} \to \overline{\mathtt{D}} \to \mathtt{Set}_d$ *such that for any* $x : \mathtt{D}\ \bar{u}$ *and* $y : \mathtt{D}\ \bar{v}$ *with* $x \prec y$*, we have* $x \not<_{\mathtt{D}} y \to \bot$*. We also define* $\bar{a} \not\leq_{\mathtt{D}} \bar{b} := \bar{a} \not<_{\mathtt{D}} \bar{b} \times \bar{a} \not\equiv_{\mathtt{D}} \bar{b}$.

If $x : \mathtt{D}\ \bar{u}$ and $y : \mathtt{D}\ \bar{v}$ then we often leave the indices implicit and write $x \not<_{\mathtt{D}} y$ and $x \not\leq_{\mathtt{D}} y$ instead of $(\bar{u}; x) \not<_{\mathtt{D}} (\bar{v}; y)$ and $(\bar{u}; x) \not\leq_{\mathtt{D}} (\bar{v}; y)$.

Example 4.19. The type $x \not<_{\mathtt{Tree}} t$ expresses that x is *not* a subtree of t. In particular, we have the following equalities:

$$\begin{aligned} x \not<_{\mathtt{Tree}} \mathtt{leaf} &= \top \\ x \not<_{\mathtt{Tree}} (\mathtt{node}\ l\ r) &= ((x \not<_{\mathtt{Tree}} l) \times (x \not\equiv_{\mathtt{Tree}} l)) \times ((x \not<_{\mathtt{Tree}} r) \times (x \not\equiv_{\mathtt{Tree}} r)) \end{aligned} \tag{4.38}$$

Construction of $\not<_{\mathtt{D}}$. We define $\not<_{\mathtt{D}}$ in terms of $\mathtt{Below}_{\mathtt{D}}$:

$$\bar{a} \not<_{\mathtt{D}} \bar{b} := \mathtt{Below}_{\mathtt{D}}\ (\lambda \bar{b}'.\ \bar{a} \not\equiv \bar{b}')\ \bar{b} \tag{4.39}$$

By definition of $\mathtt{Below}_{\mathtt{D}}$, we have a projection $\pi : (\bar{u}; x) \not<_{\mathtt{D}} (\bar{v}; y) \to (\bar{u}; x) \not\equiv_{\overline{\mathtt{D}}} (\bar{u}; x)$ whenever $x \prec y$. Filling in $\overline{\mathtt{refl}}$ for the proof of $(\bar{u}; x) \not\equiv_{\overline{\mathtt{D}}} (\bar{u}; x)$ gives us the desired proof of $x \not<_{\mathtt{D}} y \to \bot$. □

Now we can state the property that no term can be structurally smaller than itself.

Lemma 4.20 ($\mathtt{noCycle_D}$)**.** *We have a function* $\mathtt{noCycle_D} : (\bar{a}\ \bar{b} : \overline{\mathtt{D}}) \to \bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b} \to \bar{a} \not<_{\mathtt{D}} \bar{b}$.

Example 4.21. $\mathtt{noCycle_{Tree}}$ is the proof that no tree can ever be a subtree of itself, i.e. every well-typed tree is well-founded.

Construction of $\mathtt{noCycle_D}$. [1] Note that

$$x \not<_{\mathtt{D}} \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} = ((\bar{s}_1 : \Phi_{i,1}) \to x \not\leq_{\mathtt{D}} x_1\ \bar{s}_1) \times \ldots \times ((\bar{s}_{n_i} : \Phi_{i,n_i}) \to x \not\leq_{\mathtt{D}} x_{n_i}\ \bar{s}_{n_i}) \tag{4.40}$$

by definition of $\mathtt{Below_D}$ and $\not\leq_{\mathtt{D}}$. Now to construct $\mathtt{noCycle_D}$, we start by eliminating the equation $\bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b}$ using $\overline{\mathtt{J}}$, which leaves us the goal $(\bar{a} : \overline{\mathtt{D}}) \to \bar{a} \not<_{\mathtt{D}} \bar{a}$. Next we apply $\mathtt{case_D}$ with motive $\lambda\bar{a}.\ \bar{a} \not<_{\mathtt{D}} \bar{a}$, producing for each constructor $\mathtt{c_i} : \Delta_i \to (\Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1}) \to \ldots \to (\Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \mathtt{D}\ \bar{u}_i$ the subgoal

$$\begin{aligned}&(\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ &\quad (h_1 : (\bar{s}_1 : \Phi_{i,1}) \to x_1\ \bar{s}_1 \not<_{\mathtt{D}} x_1\ \bar{s}_1)\ \ldots \\ &\quad (h_{n_i} : (\bar{s}_{n_i} : \Phi_{i,n_i}) \to x_{n_i}\ \bar{s}_{n_i} \not<_{\mathtt{D}} x_{n_i}\ \bar{s}_{n_i}) \to \\ &\quad \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not<_{\mathtt{D}} \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\end{aligned} \tag{4.41}$$

To continue, we define the auxiliary types $\mathtt{Step_{i,j}}$ for $i = 1, \ldots, k$ and $j = 1, \ldots, n_i$ as follows:

$$\begin{aligned}&\mathtt{Step_{i,j}} : (\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ &\qquad (\bar{s} : \Phi_{i,j})(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{Set}_d \\ &\mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ (\bar{u}; b) = (x_j\ \bar{s}) \not<_{\mathtt{D}} b \to (\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \not\leq_{\mathtt{D}} b\end{aligned} \tag{4.42}$$

In what follows, we will construct

$$\begin{aligned}&\mathtt{step_{i,j}} : (\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ &\qquad (\bar{s} : \Phi_{i,j})(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \Phi_{i,j}\ \bar{a}\end{aligned} \tag{4.43}$$

Once this is done, we solve the subgoal (4.41) by filling in

$$\begin{aligned}&\lambda\bar{t}; x_1; \ldots; x_{n_i}; h_1; \ldots; h_{n_i}. \\ &\quad (\lambda\bar{s}_1.\ \mathtt{step_{i,1}}\ \bar{t}\ \bar{x}\ \bar{s}_1\ (\bar{v}_{i,1}; (x_1\ \bar{s}_1))\ (h_1\ \bar{s}_1)), \ldots, \\ &\quad (\lambda\bar{s}_{n_i}.\ \mathtt{step_{i,n_i}}\ \bar{t}\ \bar{x}\ \bar{s}_{n_i}\ (\bar{v}_{i,n_i}; (x_{n_i}\ \bar{s}_{n_i}))\ (h_{n_i}\ \bar{s}_{n_i}))\end{aligned} \tag{4.44}$$

So we only need to construct $\mathtt{step_{i,j}}$.

[1] Warning: understanding this construction will permanently make you a little less sane.

The construction of $\mathtt{step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s} : (\bar{a} : \overline{\mathtt{D}}) \to \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{a}$ proceeds by applying $\mathtt{elim}_{\mathtt{D}}$ with the motive $\mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}$. The new subgoals are of the form

$$\begin{array}{l}(\bar{t}' : \Delta_p)(x'_1 : \Phi_{p,1} \to \mathtt{D}\ \bar{v}'_{p,1})\ \ldots\ (x'_{n_p} : \Phi_{p,n_p} \to \mathtt{D}\ \bar{v}'_{p,n_p}) \to \\ \quad (h'_1 : (\bar{s}'_1 : \Phi_{p,1}) \to \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{v}'_{p,1}\ (x'_1\ \bar{s}'_1))\ \ldots \\ \quad (h'_{n_p} : (\bar{s}'_{n_p} : \Phi_{p,n_p}) \to \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{v}'_{p,n_p}\ (x'_{n_p}\ \bar{s}'_{n_p})) \to \\ \quad \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{u}'_p\ (\mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p})\end{array} \tag{4.45}$$

We solve them by giving:

$$\lambda \bar{t}'; x'_1; \ldots; x'_{n_p}; h'_1; \ldots; h'_{n_p}; H.\ \alpha, \beta \tag{4.46}$$

where we still have to construct

$$\alpha : (\bar{u}_i; \mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \not<_{\mathtt{D}} (\bar{u}'_p; \mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p}) \tag{4.47}$$

and

$$\beta : (\bar{u}_i; \mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \not\equiv_{\overline{\mathtt{D}}} (\bar{u}'_p; \mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p}) \tag{4.48}$$

We have $H : x_j\ \bar{s} \not<_{\mathtt{D}} \mathtt{c}_{\mathtt{p}}\ \Delta'_p\ x'_1\ \ldots\ x'_{n_p}$ or, by definition of $\not<_{\mathtt{D}}$, $H = (H_1, \ldots, H_{n_p})$ where $H_q : (\bar{s}' : \Phi'_{pq}) \to x_j\ \bar{s} \not\leq_{\mathtt{D}} x'_q\ \bar{s}'$. The construction of α reduces to the construction of components $\alpha_q : (\bar{s}' : \Phi'_{p,q}) \to \mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not\leq_{\mathtt{D}} x'_q\ \bar{s}'$. But these we can give as $\alpha_q = \lambda \bar{s}'.\ h'_q\ \bar{s}'\ (\pi_1\ (H_p\ \bar{s}'))$.

For constructing β, we assume $(\bar{u}_i; \mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \equiv_{\overline{\mathtt{D}}} (\bar{u}'_p; \mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p})$ and derive an element of $\bot$. By $\mathtt{noConf}_{\mathtt{D}}$, it suffices to consider the case where $i = p$ and $(\bar{t}; x_1; \ldots; x_{n_i}) = (\bar{t}'; x'_1, \ldots, x'_{n_p})$. But then we have $H_j\ \bar{s} : x_j\ \bar{s} \not\leq_{\mathtt{D}} x_j\ \bar{s}$, hence $\pi_2\ (H_j\ \bar{s})\ \overline{\mathtt{refl}} : \bot$. This finishes the construction of $\mathtt{noCycle}_{\mathtt{D}}$. □

4.2 Two useful techniques

In this section, we discuss two useful and general techniques that are used for the translation of definitions by pattern matching to eliminators: *basic analysis* and *specialization by unification*.

4.2.1 Basic analysis

A Ψ-elimination operator returns something of type $(\bar{t} : \Psi) \to P\ \bar{t}$ when given a motive $P : \Psi \to \mathtt{Set}_j$. However, we often need a return type where the arguments $\bar{u}$ are more specialized. For example, we may want to construct a

function of type $(k : \mathbb{N})(y : k \leq \mathtt{zero}) \to \mathtt{zero} \equiv_{\mathbb{N}} k$. Applying $\mathtt{case}_{\leq}$ directly to $y : k \leq \mathtt{zero}$ doesn't work as this leads to a loss of the information that the second index of y is $\mathtt{zero}$. McBride (2002) solves this problem by adding the constraints on the indices as additional arguments to the motive P, and filling in $\overline{\mathtt{refl}}$ as soon as the constraints are satisfied. This technique is called *basic analysis*.

Definition 4.22 (Basic analysis)**.** Let $\mathtt{elim}$ be any Ψ-elimination operator, i.e. it has a type of the form:

$$\begin{array}{l}(P : \Psi \to \mathtt{Set}_i) \to \\ (m_1 : \Delta_1 \to P\ \bar{s}_1) \ldots (m_n : \Delta_n \to P\ \bar{s}_n) \to \\ (\bar{t} : \Psi) \to P\ \bar{t}\end{array} \tag{4.49}$$

Consider some problem of type $\Delta \to T$ where $\Delta \vdash \bar{t} : \Psi$. The *basic* $\mathtt{elim}$*-analysis of* T *at* $\bar{t}$ is the term

$$\lambda m_1; \ldots; m_n; \bar{x}.\ \mathtt{elim}\ (\lambda \bar{s}.\ \Delta \to \bar{s} \equiv_{\Psi} \bar{t} \to T)\ m_1\ \ldots\ m_n\ \bar{t}\ \bar{x}\ \overline{\mathtt{refl}} \tag{4.50}$$

of type

$$(m_1 : \Delta_1 \Delta \to \bar{s}_1 \equiv_{\Psi} \bar{t} \to T) \ldots (m_n : \Delta_n \Delta \to \bar{s}_n \equiv_{\Psi} \bar{t} \to T) \to \Delta \to T \tag{4.51}$$

Basic analysis is used throughout the proof of Theorem 4.27: once with $\mathtt{rec}_{\mathtt{D}}$ for structural recursion, and once with $\mathtt{case}_{\mathtt{D}}$ for each case split.

Example 4.23. The basic $\mathtt{case}_{\leq}$-analysis of $\mathtt{zero} \equiv_{\mathbb{N}} k$ at $(k; \mathtt{zero}; y)$ has type

$$\begin{array}{l}(m_{\mathtt{lz}} : (m : \mathbb{N})(k : \mathbb{N})(y : k \leq \mathtt{zero}) \to \\ \quad (\mathtt{zero}; m; \mathtt{lz}\ m) \equiv_{(m\ n:\mathbb{N})(x:m\leq n)} (k; \mathtt{zero}; y) \to \mathtt{zero} \equiv_{\mathbb{N}} k) \\ (m_{\mathtt{ls}} : (m\ n : \mathbb{N})(x : m \leq n)(k : \mathbb{N})(y : k \leq \mathtt{zero}) \to \\ \quad (\mathtt{suc}\ m; \mathtt{suc}\ n; \mathtt{ls}\ m\ n\ x) \equiv_{(m\ n:\mathbb{N})(x:m\leq n)} (k; \mathtt{zero}; y) \to \mathtt{zero} \equiv_{\mathbb{N}} k) \to \\ (k : \mathbb{N}) \to k \leq \mathtt{zero} \to \mathtt{zero} \equiv_{\mathbb{N}} k\end{array} \tag{4.52}$$

To finish the proof that $k \leq \mathtt{zero} \to \mathtt{zero} \equiv_{\mathbb{N}} k$, we still need to give the two arguments $m_{\mathtt{lz}}$ and $m_{\mathtt{ls}}$, which we do in the next section.

4.2.2 Specialization by unification

Specialization by unification allows us to construct functions of the form $m : (\bar{x} : \Gamma)(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v}) \to T\ \bar{x}\ \bar{e}$, for example the functions of $m_{\mathtt{lz}}$ and $m_{\mathtt{ls}}$ from Example 4.23. It can be seen as a generic method of constructing an *inversion principle* (McBride, 1998b).

Definition 4.24 (Specialization by unification)**.** Consider a problem of the form $m : (\bar{x} : \Gamma)(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to T\ \bar{x}\ \bar{e}$ and suppose that unification of $\bar{u}$ with $\bar{v}$ with Γ as flexible variables succeeds either positively or negatively, then we construct the function m:

- In case the unification succeeds positively with most general unifier $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$, then we define

$$m\ \bar{x}\ \bar{e} = \overline{\texttt{subst}}\ T\ (\texttt{isLinv}\ f\ \bar{x}\ \bar{e})\ (m^s\ (f\ \bar{x}\ \bar{e})) \tag{4.53}$$

  with the new subgoal of constructing $m^s : (\bar{x}' : \Gamma') \to T\ (\texttt{linv}\ f\ \bar{x}')$.

- In case the unification succeeds negatively with disunifier $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$, then we have

$$m\ \bar{x}\ \bar{e} = \texttt{elim}_\bot\ (T\ \bar{x}\ \bar{e})\ (f\ \bar{x}\ \bar{e}) \tag{4.54}$$

  with no additional assumptions.

Example 4.25. We apply specialization by unification to construct the methods $m_{\texttt{lz}}$ and $m_{\texttt{ls}}$ from Example 4.23.

In case of $m_{\texttt{lz}}$, unification of $\bar{u} = (\texttt{zero}; m; \texttt{lz}\ m)$ with $\bar{v} = (k; \texttt{zero}; y)$ in the context $\Gamma = (m : \mathbb{N})(k : \mathbb{N})(y : k \leq \texttt{zero})$ results in a positive success with most general unifier $f : \Gamma(\bar{u} \equiv_{(m\ n:\mathbb{N})(x:m\leq n)} \bar{v}) \simeq ()$ with $f^{-1}\ () = (\texttt{zero}; \texttt{zero}; \texttt{lz}\ \texttt{zero}; \overline{\texttt{refl}})$, so specialization by unification gives us the function $m_{\texttt{lz}}$ on the condition we can construct ${m_{\texttt{lz}}}^s : \texttt{zero} \equiv_\mathbb{N} \texttt{zero}$, which we can do easily as ${m_{\texttt{lz}}}^s = \texttt{refl}$.

For $m_{\texttt{ls}}$, unification of $\bar{u} = (\texttt{suc}\ m; \texttt{suc}\ n; \texttt{ls}\ m\ n\ x)$ with $\bar{v} = (k; \texttt{zero}; y)$ results in a negative success, so specialization by unification gives us the function $m_{\texttt{ls}}$ without any additional assumptions.

When applying specialization by unification to construct a function $m : (\bar{x} : \Gamma) \to (\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to T\ \bar{x}\ \bar{e}$ from the subgoal $m^s : (\bar{x}' : \Gamma') \to T\ (\texttt{linv}\ f\ \bar{x}')$ we expect m to have 'the same' computational behaviour as m^s in case the equations $\bar{u} \equiv_\Delta \bar{v}$ are actually satisfied. This is the content of the following lemma.

Lemma 4.26. *If f is a strong unifier (Definition 3.45), then the function m constructed through specialization by unification satisfies the definitional equality $m\ (f^{-1}\ \bar{x}') = m^s\ \bar{x}'$ for any $\bar{x}' : \Gamma'$.*

Proof. Remember that for a strong unifier f, $\texttt{linv}\ f$ and $\texttt{rinv}\ f$ are definitionally equal, so it is fine to write f^{-1} here. By the first property of a strong unifier, we have $f^{-1}\ \bar{x}' = (\bar{s}; \overline{\texttt{refl}})$ for some $\bar{s} : \Gamma$. By the third

property, this implies that $\texttt{isLinv}\ f\ (f^{-1}\ \bar{x}') = \overline{\texttt{refl}}$. By the fourth property, we also have $f\ (f^{-1}\ \bar{x}') = \bar{x}'$. So we have

$$\begin{aligned} m\ (f^{-1}\ \bar{x}') &= \overline{\texttt{subst}}\ T\ (\texttt{isLinv}\ f\ (f^{-1}\ \bar{x}'))\ (m^s\ (f\ (f^{-1}\ \bar{x}'))) \\ &= \overline{\texttt{subst}}\ T\ \overline{\texttt{refl}}\ (m^s\ \bar{x}') \\ &= m^s\ \bar{x}' \end{aligned} \tag{4.55}$$

□

The fact that $m\ (\bar{t}\ \overline{\texttt{refl}})$ evaluates to $m^s\ (f\ \bar{t}\ \overline{\texttt{refl}})$ ensures that the computational behaviour corresponds to the clause written by the user, as we will see in the next section.

4.3 From pattern matching to eliminators

In this section, we prove our main theorem showing that definitions by dependent pattern matching satisfying our criterion can be translated to type theory with universes and inductive families, without using K or any other axioms.

Theorem 4.27. *Let* $\texttt{f} : (\bar{t} : \Delta) \to T$ *be a function given by a valid case tree, adhering to the following two restrictions:*

- *For each case split in the case tree on a variable* $x : \texttt{D}\ \bar{u}$ *and for each constructor* $\texttt{c} : \Delta_\texttt{c} \to \texttt{D}\ \bar{v}$ *of* D*, we have either a positive unifier of type* $\Gamma(\bar{e} : \bar{u} \equiv_\Phi \bar{v}) \simeq \Gamma'$ *or a negative unifier of type* $\Gamma(\bar{e} : \bar{u} \equiv_\Phi \bar{v}) \simeq \bot$.
- *When checking termination of a function by pattern matching, the type of the argument on which the function is structurally recursive is of the form* $\texttt{D}\ \bar{u}$ *where* D *is a datatype.*

Then we can construct a term $\texttt{f'} : (\bar{t} : \Delta) \to T$ *constructed from eliminators only.*

Example 4.28. As an example, we translate the definition of antisym (Example 1.12) to eliminators. We start from the case tree for antisym given in Figure 2.9. We follow the general procedure for translating a case tree to eliminators, so the result will be more complex than the version of antisym given in Example 1.19.

We start with the goal of constructing a function

$$\texttt{antisym'} : (m\ n : \mathbb{N})(x : m \leq n)(y : n \leq m) \to m \equiv_\mathbb{N} n \tag{4.56}$$

First, we choose an argument on which the definition is structurally recursive. Since `antisym` is structurally recursive on all four arguments, we arbitrarily choose the third one. By basic $\mathtt{rec}_{\leq}$-analysis of $m \equiv_{\mathbb{N}} n$ at $(m; n; x)$, it is sufficient to construct a function

$$\begin{array}{rcl} \mathtt{antisym}^r & : & (m'\ n' : \mathbb{N})(x' : m' \leq n') \to \mathtt{Below}_{\leq}\ P\ m'\ n'\ x' \\ & \to & (m\ n : \mathbb{N})(x : m \leq n)(y : n \leq m) \\ & \to & (m'; n'; x') \equiv_{\overline{\leq}} (m; n; x) \to m \equiv_{\mathbb{N}} n \end{array} \tag{4.57}$$

where $P\ m'\ n'\ x' = (m\ n : \mathbb{N})(x : m \leq n)(y : n \leq m) \to (m'; n'; x') \equiv_{\overline{\leq}} (m; n; x) \to m \equiv_{\mathbb{N}} n$. By applying $\overline{\mathrm{J}}$ to the equations $m'; n'; x' \equiv_{\overline{\leq}} m; n; x$, it suffices to construct a function

$$\mathtt{antisym}^s : (m\ n : \mathbb{N})(x : m \leq n)(y : n \leq m) \to \mathtt{Below}_{\leq}\ P\ m\ n\ x \to m \equiv_{\mathbb{N}} n \tag{4.58}$$

Compared to the original goal, we have gained the ability to make recursive calls by using the argument of type $\mathtt{Below}_{\leq}\ P\ m\ nx$.

Now we follow the case tree of `antisym`. By basic $\mathtt{case}_{\leq}$-analysis of $\mathtt{Below}_{\leq}\ P\ m\ n\ x \to m \equiv_{\mathbb{N}} n$ at $(m; n; x)$, it suffices to construct two functions:

$$\begin{array}{rcl} m_{\mathtt{lz}} & : & (k\ m\ n : \mathbb{N})(x : m \leq n)(y : n \leq m) \\ & \to & (\mathtt{zero}; k; \mathtt{lz}\ k) \equiv_{\overline{\leq}} (m; n; x) \\ & \to & \mathtt{Below}_{\leq}\ P\ m\ nx \to m \equiv_{\mathbb{N}} n \end{array} \tag{4.59}$$

and

$$\begin{array}{rcl} m_{\mathtt{ls}} & : & (k\ l : \mathbb{N})(w : k \leq l)(m\ n : \mathbb{N})(x : m \leq n)(y : n \leq m) \\ & \to & (\mathtt{suc}\ k; \mathtt{suc}\ l; \mathtt{ls}\ k\ l\ w) \equiv_{\overline{\leq}} (m; n; x) \\ & \to & \mathtt{Below}_{\leq}\ P\ m\ nx \to m \equiv_{\mathbb{N}} n \end{array} \tag{4.60}$$

For both goals, we apply specialization by unification on the equations $(\mathtt{zero}; k; \mathtt{lz}\ k) \equiv_{\overline{\leq}} (m; n; x)$ and $(\mathtt{suc}\ k; \mathtt{suc}\ l; \mathtt{ls}\ k\ l\ w) \equiv_{\overline{\leq}} (m; n; x)$ respectively. Unification succeeds positively in both cases, leaving us with the two new goals of constructing

$$m^s_{\mathtt{lz}} : (n : \mathbb{N})(y : n \leq \mathtt{zero}) \to \mathtt{Below}_{\leq}\ P\ \mathtt{zero}\ n\ (\mathtt{lz}\ n) \to \mathtt{zero} \equiv_{\mathbb{N}} n \tag{4.61}$$

and

$$\begin{array}{rcl} m^s_{\mathtt{ls}} & : & (k\ l : \mathbb{N})(w : k \leq l)(y : \mathtt{suc}\ l \leq \mathtt{suc}\ k) \\ & \to & \mathtt{Below}_{\leq}\ P\ (\mathtt{suc}\ k)\ (\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ w) \\ & \to & \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l \end{array} \tag{4.62}$$

We handle each of these two goals in turn.

- To construct $m^s_{\mathtt{lz}}$, we again apply basic $\mathtt{case}_{\leq}$-analysis of

$$\mathtt{Below}_{\leq}\ P\ \mathtt{zero}\ n\ (\mathtt{lz}\ n) \to \mathtt{zero} \equiv_{\mathbb{N}} n \tag{4.63}$$

at $(n; \mathtt{zero}; y)$, resulting in two subgoals

$$\begin{aligned} m_{\mathtt{lzlz}}\ :\ & (k'\ n : \mathbb{N})(y : n \leq \mathtt{zero}) \to (\mathtt{zero}; k'; \mathtt{lz}\ k') \equiv_{\overline{\leq}} (n; \mathtt{zero}; y) \\ \to\ & \mathtt{Below}_{\leq}\ P\ \mathtt{zero}\ n\ (\mathtt{lz}\ n) \to \mathtt{zero} \equiv_{\mathbb{N}} n \end{aligned} \tag{4.64}$$

and

$$\begin{aligned} m_{\mathtt{lzls}}\ :\ & (k'\ l' : \mathbb{N})(w' : k' \leq l')(n : \mathbb{N})(y : n \leq \mathtt{zero}) \\ \to\ & (\mathtt{suc}\ k'; \mathtt{suc}\ l'; \mathtt{ls}\ k'\ l'\ w') \equiv_{\overline{\leq}} (n; \mathtt{zero}; y) \\ \to\ & \mathtt{Below}_{\leq}\ P\ \mathtt{zero}\ n\ (\mathtt{lz}\ n) \to \mathtt{zero} \equiv_{\mathbb{N}} n \end{aligned} \tag{4.65}$$

We apply specialization by unification on the equations in both subgoals. The first one ends in a positive success, resulting in the subgoal

$$m^s_{\mathtt{lzlz}} : \mathtt{Below}_{\leq}\ P\ \mathtt{zero}\ \mathtt{zero}\ (\mathtt{lz}\ zero) \to \mathtt{zero} \equiv_{\mathbb{N}} \mathtt{zero} \tag{4.66}$$

which we solve by giving the right-hand side of the first clause: $m_{\mathtt{lzlz}} = \lambda\_.\ \mathtt{refl}$. The second one ends in a negative success, so it is solved without any new subgoals.

- To construct $m^s_{\mathtt{ls}}$, we once more apply basic $\mathtt{case}_{\leq}$-analysis of

$$\mathtt{Below}_{\leq}\ P\ (\mathtt{suc}\ k)\ (\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ w) \to \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l \tag{4.67}$$

at $(\mathtt{suc}\ l; \mathtt{suc}\ k\ y)$ resulting in two subgoals

$$\begin{aligned} m_{\mathtt{lslz}}\ :\ & (k'\ k\ l : \mathbb{N})(w : k \leq l)(y : \mathtt{suc}\ l \leq \mathtt{suc}\ k) \\ \to\ & (\mathtt{zero}; k'; \mathtt{lz}\ k') \equiv_{\overline{\leq}} (\mathtt{suc}\ l; \mathtt{suc}\ k\ y) \\ \to\ & \mathtt{Below}_{\leq}\ P\ (\mathtt{suc}\ k)\ (\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ w) \\ \to\ & \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l \end{aligned} \tag{4.68}$$

and

$$\begin{aligned} m_{\mathtt{lsls}}\ :\ & (k'\ l' : \mathbb{N})(w' : k' \leq l')(k\ l : \mathbb{N})(w : k \leq l)(y : \mathtt{suc}\ l \leq \mathtt{suc}\ k) \\ \to\ & (\mathtt{suc}\ k'; \mathtt{suc}\ l'; \mathtt{ls}\ k'\ l'\ w') \equiv_{\overline{\leq}} (\mathtt{suc}\ l; \mathtt{suc}\ k; y) \\ \to\ & \mathtt{Below}_{\leq}\ P\ (\mathtt{suc}\ k)\ (\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ w) \\ \to\ & \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l \end{aligned} \tag{4.69}$$

We apply specialization on the equations in both subgoals. The first one ends in a negative success, so it is solved without any subgoals. The

second one ends in a positive success, allowing us to construct $m_{\texttt{lsls}}$ from the subgoal

$$\begin{aligned} m^s_{\texttt{lsls}} \;:\; & (k\ l : \mathbb{N})(w : k \leq l)(w' : l \leq k) \\ \to\ & \texttt{Below}_{\leq}\ P\ (\texttt{suc}\ k)\ (\texttt{suc}\ l)\ (\texttt{ls}\ k\ l\ w) \\ \to\ & \texttt{suc}\ k \equiv_{\mathbb{N}} \texttt{suc}\ l \end{aligned} \tag{4.70}$$

Because $w \prec \texttt{ls}\ k\ l\ w$, we get by definition of $\texttt{Below}_{\leq}$ a projection $\pi : \texttt{Below}_{\leq}\ P\ (\texttt{suc}\ k)\ (\texttt{suc}\ l)\ (\texttt{ls}\ k\ l\ w) \to P\ k\ l\ w$. We use this projection to translate the recursive call in the definition of `antisym`. More specifically, we define $m^s_{\texttt{lsls}} = \lambda k; l; w; w'; H.\ \texttt{cong}\ \texttt{suc}\ (\pi\ H\ k\ l\ w\ w'\ \overline{\texttt{refl}})$.

There are no more subgoals to fill in, so this finishes the construction of the translated function `antisym'`.

We now construct the translated function in general.

Construction. Let $\texttt{f} : (\bar{t} : \Delta) \to T$ be the function given by a structurally recursive case tree we want to translate to eliminators. Then $\texttt{f}$ is structurally recursive on some $t_j : \texttt{D}\ \bar{v}$, the jth variable in Δ, where $\texttt{D}$ is a datatype. The basic $\texttt{rec}_{\texttt{D}}$-analysis of T at $(\bar{v}; t_j)$ allows us to construct $\texttt{f'}$ from a function $\texttt{f}^r$ of type

$$\texttt{f}^r : (\bar{x} : \overline{\texttt{D}}) \to \texttt{Below}_{\texttt{D}}\ P\ \bar{x} \to (\bar{t} : \Delta) \to \bar{x} \equiv_{\overline{\texttt{D}}} (\bar{v}; t_j) \to T \tag{4.71}$$

where $P : \overline{\texttt{D}} \to \texttt{Set}_i$ is defined as $P\ \bar{x} = (\bar{t} : \Delta) \to \bar{x} \equiv_{\overline{\texttt{D}}} (\bar{v}; t_j) \to T$. More explicitly, we define $\texttt{f'}$ by $\texttt{f'}\ \bar{t} = \texttt{rec}_{\texttt{D}}\ P\ \texttt{f}^r\ (\bar{v}; t_j)\ \bar{t}\ \overline{\texttt{refl}}$.

To construct $\texttt{f}^r$, we first move the argument of type $\texttt{Below}_{\texttt{D}}\ P\ \bar{x}$ to after $(\bar{t} : \Delta)$ and then apply $\overline{\texttt{J}}$ on the equations $\bar{x} \equiv_{\overline{\texttt{D}}} (\bar{v}; t_j)$. This simplifies the goal to constructing $\texttt{f}^s$ of type

$$\texttt{f}^s : (\bar{t} : \Delta) \to \texttt{Below}_{\texttt{D}}\ P\ \bar{v}\ t_j \to T \tag{4.72}$$

Note that $\texttt{f}^s$ may make 'recursive calls' on arguments structurally smaller than t_j using its argument of type $\texttt{Below}_{\texttt{D}}\ P\ \bar{v}\ t_j$.

To construct $\texttt{f}^s$, we proceed by induction on the structure of $\texttt{f}$'s case tree. So suppose that we have arrived at some node with label $[\Theta]\bar{p}$. Then we construct $m : \Theta \to \texttt{Below}_{\texttt{D}}\ P\ (\bar{v}; t_j)\tau \to T\tau$ where $\tau = [\Delta \mapsto \lceil \bar{p} \rceil]$. At the root node, we have $\Theta = \Delta$ and $\bar{p} = \bar{x}$ and $m = \texttt{f}^s$. There are two cases:

Internal node. In this case, the telescope is split on some variable y where $\Theta = \Theta_1(y : \mathtt{D}'\ \bar{v}_y)\Theta_2$ and $\mathtt{D}'$ is an inductive family. The basic $\mathtt{case}_{\mathtt{D}'}$-analysis of $\mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau \to T\tau$ at $(\bar{v}_y; y)$ has type

$$\begin{aligned} &\dots \\ \to\ &(m_{\mathtt{c}} : (\bar{s} : \Delta_{\mathtt{c}}) \to \Theta \to (\bar{u}_s; \mathtt{c}\ \bar{s}) \equiv_{\overline{\mathtt{D}}} (\bar{v}_y; y) \to \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau \to T\tau) \\ \to\ &\dots \\ \to\ &\Theta \to \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau \to T\tau \end{aligned} \tag{4.73}$$

where there is one method $m_{\mathtt{c}}$ for each constructor $\mathtt{c} : (\bar{s} : \Delta_{\mathtt{c}}) \to \mathtt{D}'\ \bar{u}_s$.

To construct the methods $m_{\mathtt{c}}$, we apply specialization by unification on the equations $(\bar{u}_s; \mathtt{c}\ \bar{s}) \equiv_{\overline{\mathtt{D}}} (\bar{v}_y; y)$, which succeeds either positively or negatively by the definition of a valid case tree (Definition 2.20). For each $\mathtt{c}$ with a positive success, we have to deliver a

$$m^s_{\mathtt{c}} : \Theta' \to \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau\sigma \to T\tau\sigma \tag{4.74}$$

where $f : (\bar{s} : \Delta_{\mathtt{c}})\Theta(\bar{e} : (\bar{u}_s; \mathtt{c}\ \bar{s}) \equiv_{\overline{\mathtt{D}}} (\bar{v}_y; y)) \simeq \Theta'$ is the most general unifier found by unification and σ is given by restricting the codomain of f^{-1} to Θ. The inductive hypothesis for the subtree corresponding to the constructor $\mathtt{c}$ gives us exactly such a function. For each $\mathtt{c}$ with a negative success, we get the function $m_{\mathtt{c}}$ without any additional hypotheses.

Leaf node. At each leaf node, we have the right-hand side $\Delta \vdash e : T\tau$. We want to instantiate $m = \lambda\bar{s}; H.\ e$, but e may still contain recursive calls to $\mathtt{f}$. We first have to replace these recursive calls by appropriate calls to $H : \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau$. So consider a recursive call $\mathtt{f}\ \bar{r}$ in e. Since $\mathtt{f}$ is structurally recursive on its jth argument, we have $r_j \prec \lceil p_j \rceil$ where $r_j : \mathtt{D}\ \bar{w}$. By construction of $\mathtt{Below}_{\mathtt{D}}$, we have a projection π such that $\pi\ H : (\bar{t} : \Delta) \to (\bar{w}; r_j) \equiv_{\overline{\mathtt{D}}} (\bar{v}; t_j) \to T$. Hence we define e' by replacing $\mathtt{f}\ \bar{r}$ by $\pi\ H\ \bar{r}\ \overline{\mathtt{refl}} : T[\Delta \mapsto \bar{r}]$ in e, and take $m = \lambda\bar{s}; H.\ e'$.

By induction, we now have the required $\mathtt{f}^s : \mathtt{Below}_{\mathtt{D}}\ P\ \bar{v}\ x \to (\bar{t} : \Delta) \to T$, thus finishing the construction of $\mathtt{f}$'. □

4.3.1 Computational behaviour of the translated function

What's left to prove is that when $\mathtt{f}\ \bar{t} = u$, we also have $\mathtt{f'}\ \bar{t} = \{u\}^{\mathtt{f} \mapsto \mathtt{f'}}$.

Theorem 4.29. *Let $\mathtt{f} : (\bar{t} : \Delta) \to T$ be a function satisfying the requirements of Theorem 4.27 and let $\mathtt{f'}$ be the translated version of $\mathtt{f}$ given by that theorem. Suppose moreover that all the positive unifiers used in constructing the case tree*

of `f` *are strong unifiers. Define* $\{e\}^{\mathtt{f}\mapsto\mathtt{f'}}$ *by replacing all occurrences of* `f` *by* `f'` *in* e*. Then* `f'` *satisfies* $\mathtt{f'}\ \bar{t} = \{u\}^{\mathtt{f}\mapsto\mathtt{f'}}$ *whenever* $\mathtt{f}\ \bar{t} = u$*, i.e. it has the same reduction behaviour as* `f`*.*

Proof. First, for any $\bar{t} : \Delta$ we have

$$\begin{aligned}\mathtt{f'}\ \bar{t} &= \mathtt{rec_D}\ P\ \mathtt{f}^r\ (\bar{v}; t_j)\ \bar{t}\ \overline{\mathtt{refl}} \\ &= \mathtt{f}^r\ (\bar{v}; t_j)\ (\mathtt{below_D}\ P\ \mathtt{f}^r\ (\bar{v}; t_j))\ \bar{t}\ \overline{\mathtt{refl}} \\ &= \mathtt{f}^s\ \bar{t}\ (\mathtt{below_D}\ P\ \mathtt{f}^r\ (\bar{v}; t_j))\end{aligned} \tag{4.75}$$

by definition of $\mathtt{rec_D}$ and $\mathtt{f}^r$.

Now consider a clause

$$\mathtt{f}\ \bar{p} = e \tag{4.76}$$

with pattern variables Θ at a leaf node of `f`'s case tree. By construction of the case tree, we have

$$\bar{p} = \bar{x}_0[\Theta_0 \mapsto (f_1^{-1}\ \bar{x}_1)|_{\Theta_0}] \cdots [\Theta_{n-1} \mapsto (f_n^{-1}\ \bar{x}_n)|_{\Theta_{n-1}}] \tag{4.77}$$

where $f_i : (\bar{s} : \Delta_{\mathtt{c}})\Theta_i(\bar{e} : (\bar{u}_s; \mathtt{c}\ \bar{s}) \equiv_{\bar{\mathtt{D}}} (\bar{v}_y; y)) \simeq \Theta_{i+1}$ is the most general unifier computed at the ith node on the path from the root to the leaf corresponding to this clause, $\bar{x}_i$ are pattern variables of type Θ_i, $\Theta_0 = \Delta$ and $\Theta_n = \Theta$.

To determine the computational behaviour of $\mathtt{f}^s$ applied to $\lceil\bar{p}\rceil$, we show by induction on i that $\mathtt{f}^s\ \lceil\bar{p}_i\rceil = m_i\ \bar{x}_i$ where m_i is the function m constructed at the ith node on the path and

$$\bar{p}_i = \bar{x}_0[\Theta_0 \mapsto (f_1^{-1}\ \bar{x}_1)|_{\Theta_0}] \cdots [\Theta_{i-1} \mapsto (f_i^{-1}\ \bar{x}_i)|_{\Theta_{i-1}}] \tag{4.78}$$

is the label at that node.

- In the base case we have $i = 0$, $m_i = \mathtt{f}^s$, and $\bar{p}_0 = \bar{x}_0 : \Delta$, so we have $\mathtt{f}^s\ \lceil\bar{p}_0\rceil = m_i\ \bar{x}_0$ without taking any evaluation steps.
- For the induction step we have $\bar{p}_{i+1} = \bar{p}_i[\Theta_i \mapsto (f_{i+1}^{-1}\ \bar{x}_{i+1})|_{\Theta_i}]$, hence $\mathtt{f}^s\ \lceil\bar{p}_{i+1}\rceil = m_i\ \bar{x}_i[\Theta_i \mapsto (f_{i+1}^{-1}\ \bar{x}_{i+1})|_{\Theta_i}] = m_i\ ((f_{i+1}^{-1}\ \bar{x}_{i+1})|_{\Theta_i})$ by the induction hypothesis. So it is sufficient to prove that

  $$m_i\ ((f_{i+1}^{-1}\ \bar{x}_{i+1})|_{\Theta_i}) = m_{i+1}\ \bar{x}_{i+1} \tag{4.79}$$

  By definition of m_i, we have

  $$\begin{aligned}m_i\ \bar{x} = \mathtt{case_D}\ &(\lambda\bar{z}.\ \Theta \to \bar{z} \equiv_{\overline{\mathtt{D'}}} (\bar{v}_y; y) \to \mathtt{Below_D}\ P\ (\bar{v}; t_j)\tau \to T\tau) \\ &\ldots\ m_{\mathtt{c}}\ \ldots\ (\bar{v}_y; y)\ \bar{x}\ \overline{\mathtt{refl}}\end{aligned} \tag{4.80}$$

where $m_{\mathtt{c}}$ is defined by specialization by unification from ${m_{\mathtt{c}}}^s = m_{i+1}$. By construction of the strong unifier f_{i+1}, we have that $(f_{i+1}^{-1}\ \bar{x}_{i+1})$ is of the form $(\bar{s}; (\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2); \overline{\mathtt{refl}})$, so we have

$$\begin{aligned} m_i\ ((f_{i+1}^{-1}\ \bar{x}_{i+1})|_{\Theta_i}) &= m_i\ \bar{t}_1\ (\mathtt{c}\ \bar{s})\ \bar{t}_2 \\ &= m_{\mathtt{c}}\ \bar{s}\ (\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2)\ \overline{\mathtt{refl}} \\ &= m_{\mathtt{c}}\ (f_{i+1}^{-1}\ \bar{x}_{i+1}) \\ &= m_{i+1}\ \bar{x}_{i+1} \end{aligned} \tag{4.81}$$

where the final equality follows from Lemma 4.26 and the fact that $m_{\mathtt{c}}^s = m_{i+1}$ by definition.

This completes the induction, so in particular we have $\mathtt{f}^s\ \lceil \bar{p} \rceil = \mathtt{f}^s\ \lceil \bar{p}_n \rceil = m_n\ \bar{x}_n$.

For each recursive call $\mathtt{f}\ \bar{r}$ in e, we apply Lemma 4.13 to get:

$$\begin{aligned} &\pi\ (\mathtt{below}_{\mathtt{D}}\ P\ \mathtt{f}^r\ (\bar{v}; t_j))\ \bar{r}\ \overline{\mathtt{refl}} \\ &\quad = \mathtt{f}^r\ (\bar{w}; r_j)\ (\mathtt{below}_{\mathtt{D}}\ P\ \mathtt{f}^r\ (\bar{w}; r_j))\ \bar{r}\ \overline{\mathtt{refl}} \\ &\quad = \mathtt{f}^s\ \bar{r}\ (\mathtt{below}_{\mathtt{D}}\ P\ \mathtt{f}^r\ (\bar{w}; r_j)) = \mathtt{f'}\ \bar{r} \end{aligned} \tag{4.82}$$

Finally, we are now ready to compute $\mathtt{f'}\ \lceil \bar{p} \rceil$:

$$\begin{aligned} \mathtt{f'}\ \lceil \bar{p} \rceil &= \mathtt{f}^s\ \lceil \bar{p} \rceil\ (\mathtt{below}_{\mathtt{D}}\ P\ \mathtt{f}^r\ \bar{u}\ \lceil p_j \rceil) \\ &= m_n\ \bar{x}_n\ (\mathtt{below}_{\mathtt{D}}\ P\ \mathtt{f}^r\ \bar{u}\ \lceil p_j \rceil) \\ &= e'[H \mapsto \mathtt{below}_{\mathtt{D}}\ P\ \mathtt{f}^r\ \bar{u}\ \lceil p_j \rceil] \\ &= \{e\}^{\mathtt{f} \mapsto \mathtt{f'}} \end{aligned} \tag{4.83}$$

Hence we conclude that whenever $\mathtt{f}\ \bar{t} = u$, we also have $\mathtt{f'}\ \bar{t} = \{u\}^{\mathtt{f} \mapsto \mathtt{f'}}$, as we wanted to prove. □

Chapter 5

Conclusion

Look at me still talking when there's Science to do.

— GLaDOS (2007)

Dependent pattern matching is a powerful yet intuitive tool for writing readable programs and proofs in a dependently typed language. This thesis shows how to implement it without relying on any axioms beyond standard dependent type theory. This allows everyone to use dependent pattern matching in their favourite variant of type theory, be it homotopy type theory, syntactic type theory, classical logic, or something else.

During the development of the proof-relevant unification algorithm that is used for case analysis, we present unification rules as terms internal to the object type theory. Thus the type system itself enforces the correctness of these unification rules. Moreover, this lets us extend the unification algorithm with new principles in a safe and modular way. For example, we showed how to add two new unification rules for η-equality of record types. As another example, higher-dimensional unification augments the power of the injectivity rule by allowing us to skip unification of forced arguments, yet would be impossible to even formulate for an untyped unification algorithm. So we use the power of dependent types to improve the state of dependently typed programming itself.

The impact of our work is most visible when you take dependently typed programming seriously: it makes it as easy to write programs on dependently typed data structures as it is on simply typed ones. It does so by offloading the mechanical part of reasoning about equality proofs to the unification algorithm, freeing you to think about the essential parts. So it allows you to write programs

with more interesting and expressive types without paying an extra price in complexity.

Having an elegant theoretical framework for unification also helped us a lot when implementing it in practice. As a result, the implementation of our algorithm for Agda has become cleaner, more robust, and more easily extensible. We hope this will also be the case for implementers of other dependently typed languages, as it has already been for the Lean theorem prover and the Equations package for Coq.

The path forward for dependently typed programming is long, but with the push for dependent types in more mainstream languages like Haskell, the future is bright. One day, I dare to say, it will be no harder to write a provably correct dependently typed program than it is to write a simply typed one.

5.1 Discussion and future work

This thesis wouldn't be a work of science if there weren't a lot of questions left to be answered and things left to be done. In this section we list some of the questions we think could lead to interesting new work in the future.

Other applications of proof-relevant unification

In this work, we focus on one application of proof-relevant unification, namely specialization by unification and its role in the compilation of dependent pattern matching. However, we believe firmly that it could also be applied elsewhere, for example for metaprogramming, tactic systems, or perhaps even dependently typed logic programming.

More unification rules

It would be interesting to further explore the correspondence between unification rules and new features of type theory. For example, it seems that E-unification (unification modulo a set of equations) could correspond to new unification rules for higher inductive types from HoTT. As another example, higher-order (pattern) unification could correspond to functional extensionality as a unification rule. And since the univalence axiom is itself an equivalence, maybe it could be seen as a unification rule as well?

Custom unification rules

We can put the power of unification in the hands of the user by allowing them to define custom unification rules in the form of *hints* (Asperti et al., 2009). For example, if the user provides a proof of $(f\ x \equiv_B f\ y) \simeq (x \equiv_A y)$ for some function $f : A \to B$, then this could be used as an injectivity rule for f by the unifier. Two problems prevent us from allowing user-provided unification rules in our current Agda implementation: these instances might contain free meta-variables, and they might not be strong unification rules.

We see two possible approaches of how to allow custom unification rules to be provided by the user without breaking soundness. The first one is to actually perform the desugaring of pattern matching to eliminators, so the user-provided unification rule can be incorporated into the resulting term. The resulting desugared functions would be type-safe, but their computational behaviour would naturally depend on the computational behaviour of the user-provided unification rules. As a result, the function clauses would not hold definitionally. The second possibility is to implement a check that the user-provided unification rule is actually a strong one and that it doesn't contain any unsolved metavariables. We think it could be interesting to investigate whether such a solution would be useful in practice.

Partial unification

When the unification algorithm gets stuck at some point before solving all equations, there can still be some value in the partial unifier it has constructed so far. So it would make sense to provide this partial unifier to the user, so they can handle the remaining equations by hand. To do so, the syntax for patterns would have to be extended so that the remaining equality proofs can be bound.

Example 5.1. Let $l : \mathbb{N} \to \mathbb{N}$ be an arbitrary function, then we could allow the user to define a function $\mathtt{f} : (n : \mathbb{N}) \to \mathtt{Vec}\ A\ (l\ n) \to T$ as follows:

$$\begin{aligned} &\mathtt{f} : (m : \mathbb{N}) \to \mathtt{Vec}\ A\ (l\ m) \to T \\ &\mathtt{f}\ m\ (\mathtt{nil}[e_1]) &&= \ldots \\ &\mathtt{f}\ m\ (\mathtt{cons}[e_2]\ n\ x\ xs) &&= \ldots \end{aligned} \tag{5.1}$$

where $e_1 : l\ m \equiv_{\mathbb{N}} \mathtt{zero}$ and $e_2 : l\ m \equiv_{\mathbb{N}} \mathtt{suc}\ n$.

One difficulty here is ensuring that the clauses still hold as definitional equalities in the presence of partial unifiers.

Unifiers versus strong unifiers

Not all unification rules satisfy the definition of a strong unification rule. In particular, when we construct the deletion rule for natural numbers ($\texttt{deletion}_{\mathbb{N}} : (n \equiv_{\mathbb{N}} n) \simeq ()$) using the standard eliminator for $\mathbb{N}$, the result will not be strong. For example, we have $\texttt{isLinv deletion}_{\mathbb{N}}\ \texttt{refl} = \texttt{refl}$ when n is a closed natural number ($\texttt{zero}$, $\texttt{suc zero}$, ...) but not when it is a variable from the context.

It is possible to use non-strong unification rules like this with our unification algorithm, but of course then the resulting unifier won't be a strong one. This is for example the approach taken by the Equations package for Coq. This means that the equations given in a definition by pattern matching will hold propositionally for the translated version of the function, but not definitionally. In our own implementation, we have instead chosen to require that all unification rules are strong, so we can't use unification rules like $\texttt{deletion}_{\mathbb{N}}$ but in return we can guarantee that all clauses will hold as definitional equalities.

Unification rules beyond equivalences

You may wonder why we require every unification rule to be an equivalence. At first sight, this seems to be too strict if there are no dependencies on the equality proof in question. For example, to construct a function of type $\texttt{f} : (e : x \equiv_A x) \to B$ where B doesn't depend on e, it is sufficient to construct a value for $f\ \texttt{refl}$ of type B. Given such a value of type B, we can define $\texttt{f}\ e$ for arbitrary e to be equal that value. This would suggest we only need a function $() \to (e : x \equiv_A x)$ instead of an equivalence $(e : x \equiv_A x) \simeq ()$ when there is no dependency on e.

However, we cannot ignore the computational properties of the function $\texttt{f}$. If the user only specified the clause $\texttt{f refl} = b$, then they would certainly be surprised to see that $\texttt{f}\ e = b$ for some e different from $\texttt{refl}$. For this reason, we currently require all unification rules to be equivalences, even if this is not necessary from a purely logical point of view. This rules out the definition of $\texttt{f}$ because in the absence of K, $(e : x \equiv_A x)$ is not equivalent to the empty telescope $()$. But one could certainly imagine relaxing the unification algorithm in this way if the computational behaviour of $\texttt{f}$ is less important, for example if the return type B is a mere proposition without computational content.

A broader notion of datatypes

This thesis fixes one definition of indexed datatypes. But there are plenty other notions of datatype that are used in practice. For example, we can extend the requirement of strict positivity to allow datatypes like *rose trees*:

$$\begin{aligned}&\textbf{data}\ \texttt{Rose} : \texttt{Set}\ \textbf{where}\\ &\quad \texttt{node} : \texttt{List}\ \texttt{Rose} \to \texttt{Rose}\end{aligned} \tag{5.2}$$

Our formulation of the strict positivity requirement doesn't allow this datatype because `Rose` occurs as a parameter to `List`, but it is relatively straightforward to extend the notion of strict positivity to include definitions like this.

Other more involved forms of datatypes are inductive-recursive datatypes (Dybjer, 2000) and inductive-inductive datatypes (Forsberg and Setzer, 2010). We expect that our work could be extended to include them as well, though not without significant work to first establish analogues of the no confusion and acyclicity properties for these types.

Pattern matching on higher inductive types

Our criterion makes it possible to do pattern matching on *regular* inductive families without assuming K. But HoTT also introduces the concept of *higher inductive types*, which can have non-trivial identity proofs between their constructors. This implies that in general they do not satisfy the injectivity, disjointness, or acyclicity properties. Luckily, the translation of pattern matching to eliminators is entirely *parametric* in the actual unification transitions that are used. So to allow pattern matching in a context with higher inductive types, we should start by limiting the unification algorithm further, for example by cutting out the "no confusion" and "cycle" properties for types to which they don't apply.

As a second step, these principles can be replaced by type-specific solvers that exploit any extra structure that may be available.

Example 5.2. The interval `I` is a higher inductive type with two point constructors `0 : I` and `1 : I` and one path constructor `line : 0 ≡ᵢ 1`. We have the following equivalence:

$$\texttt{contract} : (e : \texttt{0} \equiv_{\texttt{I}} \texttt{1}) \simeq () \tag{5.3}$$

By definition we have $\texttt{contract}^{-1}\ () = \texttt{line}$, so if we use this equivalence as a unification rule, we won't get a strong unification rule as a result. Maybe it is possible to weaken this requirement a bit by not requiring `refl` as such, but

merely *some* canonical form. But this means that we also need computation rules for functions applied to higher constructors, which is still an open problem. So for now, we have to settle for a weaker kind of unification rules that don't have the proper definitional behaviour, but still produce an equivalence of the correct type.

More generally, the "no confusion" principle is similar to the encode/decode technique used by Licata and Shulman (2013) and McKinna and Forsberg (2015) to calculate the fundamental group of the circle. In particular, they also construct an equivalence between an equality/path type and a type of *codes* taking the role of our `NoConfusion` type. So it may be possible to construct a new unification rule for the circle type based on this equivalence. However, be aware that these custom unification rules can introduce additional variables, for example the rule for the circle introduces a variable of type $\mathbb{Z}$! It is not yet clear how to extend the syntax of definitions by pattern matching to deal with these variables, so future research will have to show how much of the original pattern matching algorithm can be salvaged in this setting.

Pattern matching in cubical type theory

The current version of our criterion (and the corresponding proof) are written for an Agda-like theory based on standard MLTT. In such a theory, principles such as functional extensionality or univalence can be postulated but they don't get any computational behaviour. On the other hand, a new and promising theory called *cubical* type theory gives a constructive interpretation to the univalence axiom, and hence also functional extensionality (Bezem, Coquand, and Huber, 2014; Cohen et al., 2016). In the future we would like to adapt the work in this thesis to this setting, so our criterion would become usable in languages based on cubical type theory as well.

One obstacle for this adaptation is the fact that the representation of data types in our theory (and also that of Agda, Coq, Idris, ...) is computationally incompatible with the principle of functional extensionality:

$$\begin{aligned} \texttt{funext} \;&:\; \{f\ g : (x : A) \to B\ x\} \\ &\to\; (p : (x : A) \to f\ x \equiv_{B\ x} g\ x) \to f \equiv_{(x:A)\to B\ x} g \end{aligned} \tag{5.4}$$

We give an example to illustrate the problem.[1]

[1] Thanks to Conor McBride for pointing out the problem and giving this example.

Example 5.3. Let $\texttt{Favourite} : (\mathbb{N} \to \mathbb{N}) \to \texttt{Set}$ be a data type with one constructor $\texttt{favourite} : \texttt{Favourite}\ (\lambda x.\ 0 + x)$. We can give a proof p of $(x : \mathbb{N}) \to 0 + x \equiv x + 0$, so we have $\texttt{funext}\ p : \lambda x.\ 0 + x \equiv \lambda x.\ x + 0$ and thence

$$\texttt{subst}\ \texttt{Favourite}\ (\texttt{funext}\ p)\ \texttt{favourite} : \texttt{Favourite}\ (\lambda x.\ x + 0) \tag{5.5}$$

However, there is no closed canonical form of type $\texttt{Favourite}\ (\lambda x.\ x + 0)$ so this term doesn't reduce to a canonical form. This cannot be fixed by taking the constructor itself to be the canonical form (i.e. by letting $\texttt{favourite} : \texttt{Favourite}\ (\lambda x.\ x + 0)$), as this would require the typechecker to check whether two functions are extensionally equal, which is undecidable in general.

This incompatibility could be solved by disallowing indexed data types and instead having each constructor carry explicit proofs of the constraints it imposes on the former indices. For example, `favourite` would have the internal type $(e : f \equiv (\lambda x.\ 0 + x)) \to \texttt{Favourite}\ f$. The surface-level constructor is then represented as $\texttt{favourite}\ \texttt{refl}$, while $\texttt{subst}\ \texttt{Favourite}\ (\texttt{funext}\ p)\ \texttt{favourite}$ computes to $\texttt{favourite}\ (\texttt{funext}\ p)$. With this representation of data types, the work done in this thesis is just as necessary as before (modulo some details in the final proof), since we still need unification to solve the (telescopic) equations embedded in the constructors, as well as equations between these embedded equality proofs.

Example 5.4. We illustrate this by working out Example 3.54 again for a version of the `Vec` datatype with embedded equality proofs instead of indices. Suppose $\texttt{Vec}\ A\ n$ is defined with constructors $\texttt{nil} : n \equiv_{\mathbb{N}} \texttt{zero} \to \texttt{Vec}\ A\ n$ and $\texttt{cons} : (m : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ m) \to n \equiv_{\mathbb{N}} \texttt{suc}\ m \to \texttt{Vec}\ A\ n$ and consider the unification problem:

$$(e : \texttt{cons}\ n\ x\ xs\ \texttt{refl} \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)} \texttt{cons}\ n\ y\ ys\ \texttt{refl}) \tag{5.6}$$

Since this version of the `Vec` datatype doesn't have an index, we can apply the $\texttt{injectivity}_{\texttt{cons}}$ rule to simplify this equation to:

$$\begin{aligned}&(e_1 : n \equiv_{\mathbb{N}} n)(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\texttt{Vec}\ A\ e_1} ys)\\ &(e_4 : \texttt{refl} \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ e_1} \texttt{refl})\end{aligned} \tag{5.7}$$

Now e_4 is an equation between equality proofs, much like the one we obtained in (3.53), except that the equality $(p : \texttt{suc}\ e_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{suc}\ n)$ is replaced with an equality $(e_4 : \texttt{refl} \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ e_1} \texttt{refl})$. Lemma 3.59 shows that these two types are in fact equivalent. So higher-dimensional unification problems also occur in languages without indexed datatypes, and hence that a general way to solve this kind of equations is equally useful in these languages.

Another part of our work that needs to be updated to work with cubical type theory is the definition of a strong unification rule (Definition 3.41). Similarly to how in cubical type theory J computes differently according to the shape of the type it is applied to, unification rules should also satisfy different computational behaviour according to the types of the equations. We leave the precise definition of a strong unifier in cubical type theory for future work.

Automatic translation to eliminators

One thing we noticed during the writing of this proof is how easily a small mistake can have grave impact on the soundness. For example, it was only after some time that we realized just disabling **deletion** was not enough, but that the **injectivity** rule also subtly depends on K. To increase our confidence, we should make the typechecker of our languages perform the translation from pattern matching to a core calculus in practice. This is already done in Epigram (McBride and McKinna, 2004; McBride, 2005) and in the Equations package for Coq by Sozeau (2010).

In contrast to Coq and most other dependently typed languages, Agda currently doesn't have a core calculus. This means Agda is in practice somewhat less trustworthy than Coq. On the other hand, not having to translate everything to a core language also has a number of advantages:

- It is easier to extend Agda with new features without having to worry about translating them to a core calculus. This means new ideas often make their way into Agda quicker than they can into Coq.
- It is not possible to break the abstraction barrier between the high-level language and the low-level core calculus in Agda, as there is none. In contrast, in Coq it can be the case that we have to prove some properties of the translated version of a function.
- The high-level version of a function is often more efficient to evaluate than the translated one. For example, Agda uses an optimized version of case trees to evaluate functions by pattern matching efficiently.

These are not necessarily arguments why we shouldn't do the translation to eliminators in practice, but we should keep these issues in mind when implementing such a translation. In the Equations package for example, the second issue is addressed by also generating a functional elimination scheme for each definition by pattern matching.

An appealing idea to continue this line of work is to internalize even more of the unification algorithm: not just unification problems and their solutions, but also the unification engine and unification strategy described in Section 3.1.3. This could be done for example by means of *datatype-generic programming* as described by Dagand (2013). This would increase our confidence in the translation even further and be a big step towards a verified typechecker for a dependently typed language implemented in the language itself.

Computation rules beyond the standard eliminators

Our current notion of dependent pattern matching is restricted to those definitions that can be translated to eliminators while preserving the same computation rules. This is one of the reasons why we only allow definitions that can be represented by a case tree. But there are many cases where we would want a more general notion of pattern matching. For example, we may want to define $\_ + \_$ in such a way that $\texttt{zero} + n = n = n + \texttt{zero}$ and $(\texttt{suc}\ m) + n = \texttt{suc}\ (m + n) = m + (\texttt{suc}\ n)$. This kind of pattern matching with overlapping clauses was studied by Allais, McBride, and Boutillier (2013) and also in our own previous work (Cockx et al., 2014a).

However, these kind of definitions cannot be represented faithfully by eliminators, so they fall outside of the framework studied in this thesis. In the future, it would be interesting to study whether we can construct a more powerful core language to include overlapping computation rules. For example, such a system could be based on rewrite rules (Cockx and Abel, 2016). However, future work will have to tell us what sensible set of rewrite rules can be allowed without breaking soundness of the theory.

Index

Bibliography

Andreas Abel. MiniAgda: Integrating sized and dependent types. In *Workshop on Partiality and Recursion in Interactive Theorem Provers*, PAR, 2010.

Andreas Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures*. 2011.

Andreas Abel. Injectivity of type constructors is partially back. Agda refutes excluded middle, 2015a. URL `https://github.com/agda/agda/issues/1406`. (on the Agda bug tracker).

Andreas Abel. Order of patterns matters for checking left hand sides, 2015b. URL `https://github.com/agda/agda/issues/1411`. (on the Agda bug tracker).

Andreas Abel. Circumvention of forcing analysis brings back easy proof of Fin injectivity, 2015c. URL `https://github.com/agda/agda/issues/1427`. (on the Agda bug tracker).

Andreas Abel. Eta-expanded implicit patterns are not used for instance search, 2015d. URL `https://github.com/agda/agda/issues/1613`. (on the Agda bug tracker).

Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(3):1–41, 2002.

Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: A sound and complete decision procedure, formalized. In *Workshop on Dependently-typed Programming*, 2013.

Thorsten Altenkirch. Without-K problem, 2012. URL `https://lists.chalmers.se/pipermail/agda/2012/004104.html`. On the Agda mailing list.

Hiromu Arakawa. Fullmetal alchemist, 2001.

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics*, 2009.

Lennart Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. 1985.

Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, 2001.

Bruno Barras, Pierre Corbineau, Benjamin Grégoire, Hugo Herbelin, and Jorge Luis Sacchini. A new elimination rule for the calculus of inductive constructions. In *Types for Proofs and Programs*, TYPES, 2009.

Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *Types for Proofs and Programs*, TYPES, 2014.

Pierre Boutillier. *De nouveaux outils pour Calculer avec des inductifs en Coq*. PhD thesis, Université Paris-Diderot-Paris VII, 2014.

Edwin Brady. Idris, a general purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5), 2013.

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, TYPES, 2003.

Thomas Braibant. A new Coq tactic for inversion, 2013. URL `http://gallium.inria.fr/blog/a-new-Coq-tactic-for-inversion`.

Luitzen E. G. Brouwer. On the significance of the principle of excluded middle in mathematics, especially in function theory. In *A Source Book in Mathematical Logic, 1879–1931 (1967)*. 1923.

Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 1940.

Arthur C. Clarke. Hazards of prophecy: The failure of imagination. In *Profiles of the Future: An Enquiry into the Limits of the Possible*. Victor Gollanz Ltd, 1962.

Jesper Cockx. Yet another way Agda `--without-k` is incompatible with univalence, 2014. URL `https://lists.chalmers.se/pipermail/agda/2014/006367.html`. On the Agda mailing list.

Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *Types for Proofs and Programs*, TYPES, 2016.

Jesper Cockx and Dominique Devriese. Lifting proof-relevant unification to higher dimensions. In *6th Conference on Certified Programs and Proofs*, CPP. ACM, 2017.

Jesper Cockx, Dominique Devriese, and Frank Piessens. Overlapping and order-independent patterns: Definitional equality for all. In *European Symposium on Programming*, ESOP, 2014a.

Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without K. In *19th International Conference on Functional Programming*, ICFP. ACM, 2014b.

Jesper Cockx, Dominique Devriese, and Frank Piessens. Unifiers as equivalences: proof-relevant unification of dependently typed data. In *21th International Conference on Functional Programming*, ICFP. ACM, 2016a.

Jesper Cockx, Dominique Devriese, and Frank Piessens. Eliminating dependent pattern matching without K. *Journal of Functional Programming*, 26, 2016b.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, 2016. URL `http://arxiv.org/abs/1611.02108`.

Thierry Coquand. An analysis of girard's paradox. INRIA, 1986.

Thierry Coquand. Pattern matching with dependent types. In *Types for Proofs and Programs*, TYPES, 1992.

Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in Coq. In *Types for Proofs and Programs*, TYPES. 1995.

Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 1934.

Pierre-Évariste Dagand. *A cosmology of datatypes: reusability and dependent types*. PhD thesis, University of Strathclyde, 2013.

Nils Anders Danielsson. Heterogenous equality is crippled by the Bool /= Fin 2 fix, 2010. URL `https://github.com/agda/agda/issues/292`. (on the Agda bug tracker).

Nils Anders Danielsson. The unification machinery does not respect η-equality, 2011. URL `https://github.com/agda/agda/issues/473`. (on the Agda bug tracker).

Nils Anders Danielsson. Regression in unifier, possibly related to modules and/or heterogeneous constraints, 2014. URL `https://github.com/agda/agda/issues/1071`. (on the Agda bug tracker).

Nils Anders Danielsson. Dependent pattern matching is broken, 2015. URL `https://github.com/agda/agda/issues/1435`. (on the Agda bug tracker).

N. G. de Bruijn. *The mathematical language AUTOMATH, its usage, and some of its extensions*, pages 29–61. Springer Berlin Heidelberg, Berlin, Heidelberg, 1970.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *25th International Conference on Automated Deduction*, CADE, 2015.

Antoine de Saint-Exupéry. *Terre des hommes*. Le Livre de Poche, 1939.

Gabe Dijkstra. Disunifying non-fully applied constructors is inconsistent with function extensionality, 2015. URL `https://github.com/agda/agda/issues/1497`. (on the Agda bug tracker).

Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Proceedings of the first workshop on Logical frameworks*, 1991.

Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(02):525–549, 2000.

Fredrik Nordvall Forsberg and Anton Setzer. Inductive-inductive definitions. In *International Workshop on Computer Science Logic*, pages 454–468. Springer, 2010.

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

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*. 2006.

Joseph A. Goguen. What is unification? – A categorical view of substitution, equation and solution. In *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, 1989.

Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Logic in computer science*, LICS, pages 208–212, 1994.

William A. Howard. The formulæ-as-types notion of construction. In *The Curry-Howard Isomorphism (1995)*. 1969.

Chung Kil Hur. Agda with the excluded middle is inconsistent?, 2010. URL `https://lists.chalmers.se/pipermail/agda/2010/001522.html`. On the Agda mailing list.

Robert Jordan. *The Eye of the World*, volume 1 of *The Wheel of Time*. Tor Books, 1990.

Jean-Pierre Jouannaud and Claude Kirchner. *Solving equations in abstract algebras: A rule-based survey of unification*. 1990.

Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Letter to Peter van Emde Boas, 1977.

Nicolai Kraus and Christian Sattler. Higher homotopies in a hierarchy of univalent universes. *ACM Transactions on Computational Logic (TOCL)*, 16 (2):18, 2015.

Christoph Kreitz. *The Nuprl Proof Development System, Version 5*, 2002. URL `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`.

Chin Soon Lee, Neil D Jones, and Amir M Ben-Amram. The size-change principle for program termination. In *ACM SIGPLAN Notices*, volume 36, pages 81–92. ACM, 2001.

Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *28th Symposium on Logic in Computer Science*, LICS, 2013.

Zhaohui Luo. *Computation and reasoning: a type theory for computer science*, volume 11 of *International Series of Monographs on Computer Science*. 1994.

Cyprien Mangin and Matthieu Sozeau. Equations: a tool for dependent pattern-matching. In *Workshop on Type Theory Based Tools*, TTT, 2017.

Per Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, pages 127–172. Oxford University Press, 1972.

Per Martin-Löf. *Intuitionistic type theory*. Number 1 in Studies in Proof Theory. 1984.

Conor McBride. Towards dependent pattern matching in LEGO. Unpublished, 1998a.

Conor McBride. Inverting inductively defined relations in LEGO. In *Types for Proofs and Programs*, TYPES, 1998b.

Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 2000.

Conor McBride. Elimination with a motive. In *Types for Proofs and Programs*, TYPES, 2002.

Conor McBride. First-order unification by structural recursion. *Journal of functional programming*, 13, 2003.

Conor McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, 2005.

Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

Conor McBride, Healfdene Goguen, and James McKinna. A few constructions on constructors. In *Types for Proofs and Programs*, TYPES, 2006.

James McKinna and Fredrik Nordvall Forsberg. The encode-decode method, relationally. In *Types for Proofs and Programs*, TYPES, 2015.

Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1, 1991.

Alexandre Miquel. Re: Agda with the excluded middle is inconsistent?, 2010. URL `https://lists.chalmers.se/pipermail/agda/2010/001543.html`. Proof posted by Chung-Kil Hur on the Agda mailing list.

Jean-François Monin. Proof trick: Small inversions. In *Second Coq Workshop*, 2010.

Sarah Newton. *Mindjammer — The Roleplaying Game*. Mindjammer Press Ltd, 2016.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

Ulf Norell, Andreas Abel, and Nils Anders Danielsson. Release notes for Agda 2 version 2.3.2, 2012. URL `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Version-2-3-2`.

Christine Paulin-Mohring. Inductive definitions in the System Coq - rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, TLCA, London, UK, 1993. Springer-Verlag.

Daniel Peebles. Case splitting emits hidden record patterns that should remain implicit, 2012. URL `https://github.com/agda/agda/issues/635`. (on the Agda bug tracker).

Jason Reed. Another possible without-K problem, 2013. URL `https://lists.chalmers.se/pipermail/agda/2013/005578.html`. On the Agda mailing list.

Andrés Sicard-Ramírez. `--without-K` option too restrictive?, 2013. URL `https://lists.chalmers.se/pipermail/agda/2013/005407.html`. On the Agda mailing list.

Andrés Sicard-Ramírez. The `--without-K` option generates unsolved metas, 2016. URL `https://github.com/agda/agda/issues/1775`. (on the Agda bug tracker).

Matthieu Sozeau. Equations: A dependent pattern-matching compiler. In *Interactive theorem proving*, ITP, 2010.

Thomas Streicher. Investigations into intensional type theory, 1993. Habilitation thesis, Ludwig Maximilian University of Munich.

The Agda development team. *Agda 2.5.2 documentation*, 2016. URL `http://agda.readthedocs.io/en/v2.5.2/`.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2016. URL `https://coq.inria.fr/distrib/8.6/refman/`. Version 8.6.

The Idris community. *Documentation for the Idris language*, 2017. URL `http://docs.idris-lang.org/en/v0.99.1/`. Version 0.99.1.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

Valve Corporation. Portal, 2007.

Andrea Vezzosi. Heterogeneous equality incompatible with univalence even `--without-k`, 2015. URL `https://github.com/agda/agda/issues/1408`. (on the Agda bug tracker).

Philip Wadler. Propositions as types. *Communications of the ACM*, 2015.

Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *International Conference on Functional Programming*, ICFP, 2015.

Curriculum

Jesper Cockx was born in Leuven, Belgium, in 1990.

He received his bachelor degree in mathematics from KU Leuven in 2011 and his master degree in mathematics in 2013. He wrote his master thesis on Overlapping and Order-Independent Patterns in Type Theory under the supervision of prof. Frank Piessens and dr. Dominique Devriese.

From October 2013, he is a Ph.D. student at KU Leuven at the Distrinet research group under the supervision of Frank Piessens and Dominique Devriese. His Ph.D. is funded by the Research Foundation Flanders (FWO).

List of publications

Journal articles

Jesper Cockx, Dominique Devriese, and Frank Piessens. Eliminating dependent pattern matching without K. *Journal of Functional Programming*, 26, 2016b

Papers at International Conferences and Symposia

Jesper Cockx, Dominique Devriese, and Frank Piessens. Overlapping and order-independent patterns: Definitional equality for all. In *European Symposium on Programming*, ESOP, 2014a

Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without K. In *19th International Conference on Functional Programming*, ICFP. ACM, 2014b

Jesper Cockx, Dominique Devriese, and Frank Piessens. Unifiers as equivalences: proof-relevant unification of dependently typed data. In *21th International Conference on Functional Programming*, ICFP. ACM, 2016a

Jesper Cockx and Dominique Devriese. Lifting proof-relevant unification to higher dimensions. In *6th Conference on Certified Programs and Proofs*, CPP. ACM, 2017

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMEC-DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Leuven
jesper.cockx@cs.kuleuven.be
http://distrinet.cs.kuleuven.be