Unifiers as Equivalences

Proof-Relevant Unification of Dependently Typed Data

Jesper Cockx Dominique Devriese Frank Piessens

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium.
firstname.lastname@cs.kuleuven.be

Abstract

Dependently typed languages such as Agda, Coq and Idris use a syntactic first-order unification algorithm to check definitions by dependent pattern matching. However, these algorithms don't adequately consider the types of the terms being unified, leading to various unintended results. As a consequence, they require ad hoc restrictions to preserve soundness, but this makes them very hard to prove correct, modify, or extend.

This paper proposes a framework for reasoning formally about unification in a dependently typed setting. In this framework, unification rules compute not just a unifier but also a corresponding correctness proof in the form of an *equivalence* between two sets of equations. By rephrasing the standard unification rules in a proof-relevant manner, they are guaranteed to preserve soundness of the theory. In addition, it enables us to safely add new rules that can exploit the dependencies between the types of equations.

Using our framework, we reimplemented the unification algorithm used by Agda. 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. We are convinced this will also enable the addition of new and interesting unification rules in the future, without compromising soundness along the way.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs – functional constructs, program and recursion schemes; D.3.3 [*Programming Languages*]: Language Constructs and Features – data types and structures, patterns, recursion

Keywords Unification, Type Theory, Dependent Types, Inductive Families, Agda

1. Introduction

Unification, a generic method for solving symbolic equations algorithmically, 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, type checkers for languages with dependent pattern matching (Coquand 1992) use unification to determine whether a set of constructors covers all possible cases. For example, a function f that takes an argument of type $\texttt{Vec}\ A\ (1+n)$ (i.e. a vector containing $1+n$ elements of type A) needs only be defined in the case $f\ (\texttt{cons}\ n\ x\ xs)$ (where $\texttt{cons}\ n\ x\ xs$ is the vector with head $x : A$ and tail $xs : \texttt{Vec}\ A\ n$): the case $f\ []$ is impossible because unification of 0 (the length of $[]$) with $1+n$ reports an absurdity. In the cases where unification succeeds, it can also teach us something extra about the type of the right-hand side. For example, when analyzing a variable x of type $\texttt{Vec}\ A\ n$ it tells us $n = 0$ in the case $x = []$ and $n = \texttt{suc}\ n'$ in the case $x = \texttt{cons}\ n'\ x\ xs$. This method of solving equations to either gain more information about the type of the right-hand side or to derive an absurdity is one instance of a very useful technique called *specialization by unification* (Goguen et al. 2006).

In languages that have dependent pattern matching as a primitive such as Agda (Norell 2007), the particularities of the unification rules used become crucial for the language's notion of equality. Indeed, in Agda one can match on a proof of $u \equiv_A v$ with the constructor $\texttt{refl}$ 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 uniqueness of identity proofs (UIP) by pattern matching (Coquand 1992). So it is important to have a solid theoretical understanding of unification in order to study these languages.

When dependently typed terms themselves become the subject of unification, it is possible that we encounter *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 $\Sigma_{A:\texttt{Set}} A$ with elements (A, a) packing a type A together with an element a of that type. By injectivity of the constructor $(\cdot,\cdot)$, 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 $b : B$. The question becomes then whether we can still apply the standard unification rules to heterogeneous equations, and under what conditions.

Because traditional unification algorithms only look at the syntax of the terms they are trying to unify, they cause problems when applied in a setting with heterogeneous equalities. For example, they can simplify the equation $(\texttt{Bool}, \texttt{true}) = (\texttt{Bool}, \texttt{false})$ of type $\Sigma_{A:\texttt{Set}} A$ to $\texttt{Bool} = \texttt{Bool}$ and $\texttt{true} = \texttt{false}$. Subsequently, they 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 UIP (McBride 2000). Indeed, if we have access to the univalence axiom (The Univalent Foundations Program 2013) then we can prove that $(\texttt{Bool}, \texttt{true})$ equals $(\texttt{Bool}, \texttt{false})$ of type $\Sigma_{A:\texttt{Set}} A$. On the other hand, consider the exact same unification problem $(\texttt{Bool}, \texttt{true}) = (\texttt{Bool}, \texttt{false})$, but this time the type of the equation is a non-dependent product $\texttt{Set} \times \texttt{Bool}$. In this case it *is* possible to derive an absurdity. However, a syntax-directed unification algorithm will never be able to distinguish between these two cases.

The problem isn't limited to theories that don't support UIP, either. For example, let A be an arbitrary type and `Singleton` : $A \to$ `Set` be an indexed data type with one constructor `sing` : $(x : A) \to$ `Singleton` x and consider the unification problem $($`Singleton` $s,$ `sing` $s) = ($`Singleton` $t,$ `sing` $t)$. If we allow the injectivity rule to simplify `sing` $x =$ `sing` y to $x = y$ then this problem can be solved with solution $y \mapsto x$. However, this would allow us to prove injectivity of the type constructor `Singleton`. Injectivity of type constructors is generally an undesirable property, as it is not only incompatible with univalence but also with impredicativity (Miquel 2010) and with the law of the excluded middle (Hur 2010). In particular, if we allow this application of injectivity of the `sing` constructor for $A =$ `Set` $\to$ `Set`then we can refute the law of the excluded middle.

The goal of this paper is to give a *fully typed* account of unification of dependently typed data, in order to avoid problems like the ones described above 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 `suc` : $\mathbb{N} \to \mathbb{N}$ is represented by an equivalence between the equations `suc` $m \equiv_{\mathbb{N}}$ `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. Having this evidence means our unification rules are guaranteed to be sound with respect to the type theory we're working in. This gives us a new formal criterion for the correctness of a unification rule in a dependently typed setting. Moreover, this "evidence of unification" is produced internally to the type theory, so that it is easy to incorporate unification into other parts of the language.

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 for the application we have in mind: this equivalence can be used directly for specialization by unification as used in the compiler from dependent pattern matching to eliminators by Cockx et al. (2014). This makes it also suitable for languages with a core calculus like Coq (The Coq development team 2012), Epigram (McBride 2005), Lean (de Moura et al. 2015), or (potentially) a future version of Agda.

Contributions.

- We give a 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.
- We rephrase the unification rules of Goguen et al. (2006) in this framework and show that they conform to our definition of unification rules.
- We present new unification rules for indexed families of data types that work on heterogeneous equations and can work on multiple equations at once, making them more general than the ones given by Cockx et al. (2014).
- We also describe two new unification rules that deal with eta-equality of values of record type.
- We describe how it can be useful to apply unification rules in the reverse direction in order to solve problems where the equations between the indices are not fully general.
- We reimplement the unification algorithm used by Agda (Norell 2007) for pattern matching on indexed families of data types using our new framework for unification, fixing a number of bugs in the process and making it more amenable to future extensions. This new unification algorithm will be released as part of Agda version 2.5.1, of which there is currently a release candidate available[1].

The type theory used in this paper is a version of Luo (1994)'s Unified Theory of Dependent Types (UTT), but our results should be equally applicable in other intuitionistic type theories with inductive families such as the Calculus of Inductive Constructions used by Coq. The basic rules of the type theory are summarized in Figure 1. We use Greek capitals $\Gamma, \Delta, \ldots$ for both contexts and telescopes, capitals $T, U, \ldots$ for types, and small letters $t, u, \ldots$ for terms. The simultaneous substitution of the terms $\bar{t}$ for the variables in the telescope Δ is written as $[\Delta \mapsto \bar{t}]$. Throughout the paper, we make use of concepts such as the identity type, telescope notation, and inductive families of data types. We introduce these concepts as the need for them arises.

We also use concepts from homotopy type theory (HoTT) (The Univalent Foundations Program 2013) such as an equality "lying over" another one and the concept of an *equivalence* between types. Our notation for telescopic equality is also inspired by cubical type theory (Cohen et al. 2015). However, our unification rules don't require any axioms on top of basic intuitionistic type theory (such as the univalence axiom). 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 (e.g. uniqueness of identity proofs, or the law of the excluded middle).

The rest of this paper is organized as follows. Section 2 shows how unification problems and most general unifiers can be represented internally to type theory, and gives an important use of this internal representation: specialization by unification. Section 3 continues by giving a number of examples of unification problems and their solutions, showing how our unification rules work and how they prevent the problems in untyped unification algorithms. Section 4 discusses unification rules in general, and shows how they can be used as building blocks for constructing the most general unifier. It also shows how the basic unification rules from Goguen et al. (2006) can be interpreted in our setting. By representing unifiers as terms in type theory, they also get a computational behaviour, which is discussed in Section 5. The basic unification rules are joined by a number of other rules in Section 6: η rules for record types, inverse unification rules for generalization of data type indices, and an example of a unification rule for a higher inductive type: the interval. Section 7 discusses the implementation of our ideas in the Agda type checker, and see how it compares to the previous implementation on soundness, complexity, and extensibility.

[1] See `https://lists.chalmers.se/pipermail/agda/2016/008532.html` for installation instructions.

$$\frac{}{\epsilon \textbf{ context}}\ \text{(Ctx-empty)}$$

$$\frac{\Gamma \vdash A : \mathtt{Set}_i \qquad x \notin FV(\Gamma)}{\Gamma(x : A) \textbf{ context}}\ \text{(Ctx-extend)}$$

$$\frac{\Gamma \textbf{ context} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{(Var)}$$

$$\frac{\Gamma \vdash t : A_1 \qquad \Gamma \vdash A_1 = A_2 : \mathtt{Set}_i}{\Gamma \vdash t : A_2}\ \text{(=Ty)}$$

$$\frac{\Gamma \textbf{ context}}{\Gamma \vdash \mathtt{Set}_i : \mathtt{Set}_{i+1}}\ \text{(Set)}$$

$$\frac{\Gamma \vdash A : \mathtt{Set}_i \qquad \Gamma(x : A) \vdash B : \mathtt{Set}_j}{\Gamma \vdash (x : A) \to B : \mathtt{Set}_{\max(i,j)}}\ (\Pi)$$

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

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

$$\frac{\Gamma(x : A) \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x.\, t)\ s = t[x \mapsto s] : B[x \mapsto s]}\ (\beta)$$

+ reflexivity, symmetry, transitivity and congruence rules for =

Figure 1. The core formal rules of UTT, including dependent function types $(x : A) \to B$, an infinite hierarchy of universes $\mathtt{Set}_0$ ($= \mathtt{Set}$), $\mathtt{Set}_1$, $\mathtt{Set}_2$,..., and β-equality.

Finally, Section 8 contains a discussion of related work and Section 9 concludes.

2. Unification problems and unifiers

Before we can start thinking about formally correct unification rules, we first need to specify how we represent the input and the output of the unification algorithm. The input of a unification algorithm consists of a set of equations collectively called the *unification problem*, while the output can be one of three options:

- either the algorithm finds a substitution called the *most general unifier*, and we say that the algorithm *succeeds positively*,
- or the algorithm detects an absurd equation, in which case we say the algorithm *succeeds negatively*,
- or it *fails* because there is no unification rule that applies.

The third possibility is unavoidable in general because the language we work in includes lambda expressions and defined functions.

These central concepts can be defined internally to type theory: unification problems can be represented as telescopic equalities (Sections 2.1 and 2.2), and most general unifiers can be represented as equivalences (Section 2.3). It is important to construct these most general unifiers internally to the theory so that they can be used to apply specialization by unification (Section 2.4).

2.1 Equations as types

To represent unification problems internally, we need to be able to express equality between two terms as a type. For this purpose, we use the propositional equality type $x \equiv_A y$ introduced by Martin-Löf (1984). The most basic way to prove an equality is by reflexivity: $\mathtt{refl} : x \equiv_A x$. We also have functions expressing symmetry, transitivity, and congruence:

$$\begin{aligned} \mathtt{sym} &: x \equiv_A y \to y \equiv_A x \\ \mathtt{trans} &: x \equiv_A y \to y \equiv_A z \to x \equiv_A z \\ \mathtt{cong} &: (f : A \to B) \to x \equiv_A y \to f\ x \equiv_B f\ y \end{aligned} \tag{1}$$

Terms can be substituted for equal terms by applying $\mathtt{subst}$:

$$\mathtt{subst} : (P : A \to \mathtt{Set}_i) \to x \equiv_A y \to P\ x \to P\ y \tag{2}$$

More generally, we have the J rule that also allows P to depend on the proof of $x \equiv_A y$:

$$\begin{aligned} \mathsf{J} &: (P : (y : A) \to x \equiv_A y \to \mathtt{Set}_i) \\ &\quad (p : P\ x\ \mathtt{refl})(y : A)(e : x \equiv_A y) \to P\ y\ e \end{aligned} \tag{3}$$

In fact, the J rule is powerful enough so that the functions $\mathtt{sym}$, $\mathtt{trans}$, and $\mathtt{subst}$ can readily be defined from it.

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 simply 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 axiom, which is equivalent to uniqueness of identity proofs (McBride 2000). So in a theory where we don't have access to the general K axiom (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 $\mathtt{Bool}\ {}_{\mathtt{Set}}{\cong}_{\mathtt{Set}}\ \mathtt{Bool}$ and $\mathtt{true}\ {}_{\mathtt{Bool}}{\cong}_{\mathtt{Bool}}\ \mathtt{false}$, there is no way to see whether the type of the second equation depends on the first. The example from the introduction 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 type constructors (see the $\mathtt{Singleton}$ example in the introduction).

In order 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. Concretely, if $e : s \equiv_A t$ and $P : (x : A) \to \mathtt{Set}$, then $u \equiv^e_P v$ is the type of equality proofs between $u : P\ s$ and $v : P\ t$ laying over e. Note that P is the part of the type that u and v have in common, while e shows where they differ.

There are multiple equivalent ways to define $u \equiv^e_P v$; for the sake of simplicity we use the following definition in terms of the regular homogeneous equality by substituting by e on the left:

$$u \equiv^e_P v = (\mathtt{subst}\ P\ e\ u) \equiv_{(P\ t)} v \tag{4}$$

In practice, the exact definition of $u \equiv^e_P v$ doesn't have much impact, but we prefer this one to the more symmetric alternatives because it doesn't require large eliminations or auxiliary data types.

We will often write $u \equiv_{P\ e} v$ instead of $u \equiv^e_P v$. For example, if we have $e : m \equiv_{\mathbb{N}} n$ and two vectors $u : \mathtt{Vec}\ A\ m$ and $v : \mathtt{Vec}\ A\ n$, then we may form the type $u \equiv_{\mathtt{Vec}\ A\ e} v$. This notation is inspired by cubical type theory (Cohen et al. 2015), 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.

2.2 Unification problems as telescopes

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*. A telescope is a list of typed variable bindings where each type can depend on the previous variables, expressing the fact that the type of each equation can depend on the previous equations. For example, a possible telescope is $(m : \mathbb{N})(p : m \equiv_{\mathbb{N}} \texttt{zero})$.

Telescopes can be used as the type of a list of terms. We indicate a list of terms by a bar above the letter, so we can write for example $\bar{t} = \texttt{zero};\texttt{refl} : (m : \mathbb{N})(p : m \equiv_{\mathbb{N}} \texttt{zero})$. Formally, this means that we give a semantics to a telescope Δ as an iterated sigma type $[[\Delta]]$[2]:

$$\begin{aligned} [[()]] &= \top \\ [[(x : A)\Delta]] &= \Sigma_{(x:A)}[[\Delta]] \end{aligned} \tag{5}$$

We also rely on the concept of a function between telescopes, called a *telescope mapping*. A telescope mapping $f : \Delta \to \Delta'$ is simply a function from $[[\Delta]]$ to $[[\Delta']]$. Note that a telescope mapping f can be thought of as a substitution: if we have $\Delta' \vdash u : A$, then $\Delta \vdash u[\Delta' \mapsto f\ \Delta] : A[\Delta' \mapsto f\ \Delta]$.

Now we can define telescopic equality in full generality:

Definition 1. 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 $() \equiv_{()} () = ()$ (where we write $()$ for both the empty telescope and the empty list of terms) and

$$(e;\bar{e} : s;\bar{s} \equiv_{(x:A)\Delta} t;\bar{t}) = (e : s \equiv_A t)(\bar{e} : \bar{s} \equiv^{e}_{\lambda x.\,\Delta} \bar{t}) \tag{6}$$

For example, $(e_1;e_2 : m;u \equiv_{(x:\mathbb{N})(y:\texttt{Vec}\ A\ x)} n;v)$ stands for the telescope $(e_1 : m \equiv_{\mathbb{N}} n)(e_2 : u \equiv_{\texttt{Vec}\ A\ e_1} v)$. This telescope can be thought of as a unification problem consisting of two equations e_1 and e_2.

For each $\bar{t} : \Delta$, we define $\overline{\texttt{refl}} : \bar{t} \equiv_\Delta \bar{t}$ as $\texttt{refl};\ldots;\texttt{refl}$. We also define telescopic versions of $\texttt{subst}$ and $\texttt{cong}$:

$$\begin{aligned} \overline{\texttt{subst}} &: (P : \Delta \to \texttt{Set}_i) \to \bar{u} \equiv_\Delta \bar{v} \to P\ \bar{u} \to P\ \bar{v} \\ \overline{\texttt{cong}} &: (f : \Delta \to T) \to \bar{u} \equiv_\Delta \bar{v} \to f\ \bar{u} \equiv_T f\ \bar{v} \end{aligned} \tag{7}$$

2.3 Most general 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? Suppose we have a unification problem with free variables coming from a telescope Γ (also called the *flexible variables*) and equations $\bar{u} \equiv_\Delta \bar{v}$, where Δ, $\bar{u}$, and $\bar{v}$ can depend on the variables from Γ. We could represent a unifier as just 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 2. 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'.

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

Usually, if the unification algorithm succeeds positively it doesn't output just any unifier but a *most general* one, i.e. a unifier σ such that any other unifier σ' can be written as $\sigma \circ h$ for some h. Again, we should think how to represent this concept internally.

[2] In the rest of this paper, the brackets $[[\cdot]]$ will be implicit.

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 unifiers σ', making the definition more unwieldy than necessary. Instead, we ask that σ has a right inverse τ, i.e. a telescope mapping $\tau : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to \Gamma'$ such that $\sigma \circ \tau$ is the identity function on $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. This allows us to define $h = \tau \circ \sigma'$, which gives us $\sigma \circ h = \sigma \circ \tau \circ \sigma' = \sigma'$, as we wanted. Intuitively, τ allows us to recover the values of the variables in Γ' for any values of Γ that satisfy $\bar{u} \equiv_\Delta \bar{v}$.

It is often useful to require that the function h 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 we don't require that h is unique, then there may be other most general unifiers with a non-equivalent choice of Γ'. For example, we could also have taken $\sigma' : (b' : \texttt{Bool}) \to (b : \texttt{Bool})(e : b \equiv_{\texttt{Bool}} \texttt{true})$ by simply ignoring the argument b'. To exclude solutions like this where Γ' is unnecessarily large, we require that σ also has a left inverse τ'. If we have two different functions h and h' such that $\sigma' = \sigma \circ h = \sigma' \circ h'$ then this left inverse allows us to prove that $h = \tau' \circ \sigma \circ h = \tau' \circ \sigma \circ h' = h'$, so h is unique.

In type-theoretic terms, this means that σ is an *equivalence* between Γ' and $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. In general, an equivalence $f : A \simeq B$ consists of a function $f : A \to B$ together with left and right inverses $g_1, g_2 : B \to A$ and proofs

$$\begin{aligned} \texttt{isLinv}\ f &: (x : A) \to g_1\ (f\ x) \equiv_A x \\ \texttt{isRinv}\ f &: (y : B) \to f\ (g_2\ y) \equiv_B y \end{aligned} \tag{8}$$

In general, the left and right inverses of f can be two different functions, but they are always the same for the equivalences given in this paper. In this situation, we write f^{-1} for the common value of g_1 and g_2.

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

Definition 3. 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 we define a *most general unifier* of $\bar{u}$ and $\bar{v}$ as an equivalence $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ for some telescope Γ'.

Note that the unifier $\sigma : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ corresponds to the inverse function f^{-1}.

This definition doesn't prevent us from simply choosing $\Gamma' = \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ and $f = id$. In fact, this is a valid most general unifier from a logical point of view. Current implementations of dependent pattern matching also require that $f^{-1}\ \bar{x}'$ is of the form $\bar{u};\overline{\texttt{refl}}$ for some $\bar{u} : \Gamma$, excluding this kind of 'trivial unifier'. However, this seems to be a limitation of the current implementation of indexed data types, rather that a necessary requirement on the computational behaviour of f^{-1}.

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

Definition 4. 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 we define a *disunifier* of $\bar{u}$ and $\bar{v}$ as an equivalence $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$.

Note that 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 $\texttt{elim}_\bot : (A : \texttt{Set}_i) \to \bot \to A$, expressing the principle of 'ex falso quodlibet'. So the only interesting part of the equivalence is the function f.

2.4 Specialization by unification

Specialization by unification is a very useful and powerful technique for constructing functions of the form $m : (\bar{x} : \Gamma) \rightarrow (\bar{e} : \bar{u} \equiv_{\Delta} \bar{v}) \rightarrow T\ \bar{x}\ \bar{e}$. It can be used as a generic method of constructing *inversion principles* (McBride 1998a) and is also a crucial ingredient for translating definitions by dependent pattern matching to eliminators (Goguen et al. 2006; Cockx et al. 2014). If unification for the problem $\bar{u} \equiv_{\Delta} \bar{v}$ with Γ as flexible variables succeeds either positively or negatively, then it is straightforward to construct this 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 generate a 'subgoal' of constructing a function m' of type $(\bar{x}' : \Gamma') \rightarrow T\ (f^{-1}\ \bar{x}')$. Once we have found such an m', we have $m'\ (f\ \bar{x}\ \bar{e}) : T\ (f^{-1}\ (f\ \bar{x}\ \bar{e}))$, so we can define m by
$$m\ \bar{x}\ \bar{e} = \overline{\mathtt{subst}}\ T\ (\mathtt{isLinv}\ f\ \bar{x}\ \bar{e})\ (m'\ (f\ \bar{x}\ \bar{e})) \qquad (9)$$
where $\mathtt{isLinv}\ f\ \bar{x}\ \bar{e}$ is the proof that $f^{-1}\ (f\ \bar{x}\ \bar{e}) \equiv \bar{x}\ \bar{e}$.
- In case the unification succeeds negatively with result the disunifier $f : \Gamma(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v}) \simeq \bot$, then we can define m by
$$m\ \bar{x}\ \bar{e} = \mathtt{elim}_{\bot}\ (T\ \bar{x}\ \bar{e})\ (f\ \bar{x}\ \bar{e}) \qquad (10)$$
Note that in this case no subgoal is needed.

For example, we can apply specialization by unification to construct a function $m : (k\ l : \mathbb{N}) \rightarrow \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l \rightarrow T\ k\ l$. As we will soon see in Example 1, unification succeeds positively with the most general unifier f of type $(k\ l : \mathbb{N})(e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l) \simeq (k : \mathbb{N})$ satisfying $f^{-1}\ k = k; k; \overline{\mathtt{refl}}$. So if we can provide a function $m' : (k : \mathbb{N}) \rightarrow T\ k\ k$, then specialization by unification allows us to construct the function m.

3. Examples

Before we give the general unification rules, we first want to give an idea of what unification looks like on a few concrete examples. The general theme of these examples is that it is unsound to naively apply purely syntactic rules in a dependently typed setting. We replace these syntactic rules by typed ones that are proven correct as evidenced by an equivalence. All the rules that are used in this section are specific instances of the general rules constructed in Section 4.2: solution, deletion, injectivity, conflict, cycle, and the indexed variants of the latter three in Section 4.3.

Example 1. We start with an easy example consisting of a single equation between $\mathtt{suc}\ k$ and $\mathtt{suc}\ l$. First, we simplify the equation by applying the equivalence $(e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l) \simeq (e : k \equiv_{\mathbb{N}} l)$, which is called the *injectivity* rule for $\mathtt{suc}$. Applying this rule leaves the two variables k and l unchanged. Next, we apply the *solution* rule, 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 is finished.

$$\begin{aligned} &(k\ l : \mathbb{N})(\underline{e} : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l) \\ \simeq\ &(k\ \underline{l} : \mathbb{N})(\underline{e} : k \equiv_{\mathbb{N}} l) \\ \simeq\ &(k : \mathbb{N}) \end{aligned} \qquad (11)$$

To get the substitution from $(k : \mathbb{N})$ to $(k\ l : \mathbb{N})(e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l)$ 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 $e \mapsto \mathtt{refl}$, and the injectivity rule maps $e : k \equiv_{\mathbb{N}} l$ to $\mathtt{cong}\ \mathtt{suc}\ e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l$, so the complete substitution is $k \mapsto k; k; \mathtt{refl}$ (since $\mathtt{cong}\ \mathtt{suc}\ \mathtt{refl} = \mathtt{refl}$).

Example 2. Consider the sum type $A \uplus B$ (where $A, B : \mathtt{Set}$ are arbitrary types) with two constructors $\mathtt{inj}_1 : A \rightarrow A \uplus B$ and $\mathtt{inj}_2 : B \rightarrow A \uplus B$ (also known as the $\mathtt{Either}$ type). An expression of the form $\mathtt{inj}_1\ x$ can never be equal to $\mathtt{inj}_2\ y$, so any equality between those two terms is equivalent to $\bot$:

$$(x : A)(y : B)(\underline{e} : \mathtt{inj}_1\ x \equiv_{A \uplus B} \mathtt{inj}_2\ y) \simeq \bot \qquad (12)$$

This is called the *conflict* rule between $\mathtt{inj}_1$ and $\mathtt{inj}_2$.

We should be careful not to apply the conflict rule naively, though: an equation between the constructors $\mathtt{inj}_1$ and $\mathtt{inj}_2$ is not always absurd when they are not fully applied. For example[3], let $A = B = \bot$, then $(e : \mathtt{inj}_1 \equiv_{\bot \rightarrow \bot \uplus \bot} \mathtt{inj}_2)$ is not equivalent to $\bot$. This is because when we view $\mathtt{inj}_1$ and $\mathtt{inj}_2$ as functions of type $\bot \rightarrow \bot \uplus \bot$, they coincide on all possible inputs (i.e. none). The principle of *functional extensionality* tells us that these two functions must then be equal. So if we would consider this equation to be absurd, we would prohibit ourselves from having functional extensionality in our language, nevertheless a very desirable property to have!

Example 3. We continue with a first example involving an indexed data type: $\mathtt{Vec}\ A : \mathbb{N} \rightarrow \mathtt{Set}$. Remember that its two constructors are $[] : \mathtt{Vec}\ A\ 0$ and $\mathtt{cons} : (n : \mathbb{N}) \rightarrow A \rightarrow \mathtt{Vec}\ A\ n \rightarrow \mathtt{Vec}\ A\ (\mathtt{suc}\ n)$. The injectivity rule for $\mathtt{cons}$ gives us the following equivalence:

$$\begin{aligned} &(\mathtt{suc}\ m; \mathtt{cons}\ m\ x\ xs \equiv_{\overline{\mathtt{Vec}\ A}} \mathtt{suc}\ n; \mathtt{cons}\ n\ y\ ys) \\ \simeq\ &(m; x; xs \equiv_{(n:\mathbb{N})(x:A)(xs:\mathtt{Vec}\ A\ n)} n; y; ys) \end{aligned} \qquad (13)$$

where we write $\overline{\mathtt{Vec}\ A}$ for the telescope $(n : \mathbb{N})(x : \mathtt{Vec}\ A\ n)$. Note that this rule not only simplifies the equation between the two $\mathtt{cons}$ constructors, but also simplifies the equations between the indices $\mathtt{suc}\ m$ and $\mathtt{suc}\ n$ simultaneously. The reason for this will become clear when we consider the injectivity rule in general in Section 4.3. Now let's see how this rule works in action:

$$\begin{aligned} &(m\ n : \mathbb{N})(x\ y : A)(xs : \mathtt{Vec}\ A\ m)(ys : \mathtt{Vec}\ A\ n) \\ &(\underline{e_1} : \mathtt{suc}\ m \equiv_{\mathbb{N}} \mathtt{suc}\ n) \\ &(\underline{e_2} : \mathtt{cons}\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ y\ ys) \\ \simeq\ &\begin{matrix}(\underline{m}\ n : \mathbb{N})(x\ y : A)(xs : \mathtt{Vec}\ A\ m)(ys : \mathtt{Vec}\ A\ n) \\ (\underline{e_1} : m \equiv_{\mathbb{N}} n)(\underline{e_2} : x \equiv_A y)(\underline{e_3} : xs \equiv_{\mathtt{Vec}\ A\ e_1} ys)\end{matrix} \\ \simeq\ &(n : \mathbb{N})(x : A)(xs : \mathtt{Vec}\ A\ n) \end{aligned} \qquad (14)$$

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

Note that in order to apply injectivity of $\mathtt{cons}$, the type of the equation $\mathtt{cons}\ x\ u \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ y\ v$ has to be of the form $\mathtt{Vec}\ A\ e$ where e refers to a previous equation. This implies that we cannot apply this rule as it is to an equation of the form $\mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys$ where $xs : \mathtt{Vec}\ A\ n$ and $ys : \mathtt{Vec}\ A\ n$ have the same length 'on the nose'. We discuss how to solve this deficiency in the special case where the index type satisfies the K rule (as is the case for $\mathbb{N}$) in Section 4.3 and more in general in Section 6.2.

Example 4. 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 simplified the equation $\mathtt{suc}\ m \equiv_{\mathbb{N}} \mathtt{suc}\ n$ by appealing to the $\mathtt{injectivity}_{\mathtt{suc}}$ rule. However, sometimes this simplification gives a real increase to the power of unification. For example, let $f : A \rightarrow B$ be a (possibly very complex) function, then in general there is no way to solve an equation of the form $f\ x \equiv_B f\ y$, unless we know something about f, e.g. that it is injective. Now let $\mathtt{Im}\ f : B \rightarrow \mathtt{Set}$ be a data type with one constructor $\mathtt{image} : (x : A) \rightarrow \mathtt{Im}\ f\ (f\ x)$, then

[3] This example is based on the problem described by Dijkstra (2015).

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

$$\begin{aligned}&(x\ y : A)(\underline{e_1} : f\ x \equiv_B f\ y)\\&(\underline{e_2} : \texttt{image}\ x \equiv_{\texttt{Im}\ f\ e_1} \texttt{image}\ y)\\ \simeq\ &(x\ y : A)(e : x \equiv_A y)\\ \simeq\ &(x : A)\end{aligned} \tag{15}$$

For example, having an injectivity rule that works in this way is useful when giving semantics to an embedded language (Danielsson 2015).

Contrast this example with the situation where we have the unification problem

$$\begin{aligned}&(x\ y : A)(e_1 : \texttt{Im}\ f\ (f\ x) \equiv_{\texttt{Set}} \texttt{Im}\ f\ (f\ y))\\&(e_2 : \texttt{image}\ x \equiv_{e_1} \texttt{image}\ y)\end{aligned} \tag{16}$$

Here, we are not allowed to use injectivity on the second equation since its type is not a data type but a variable. Note that 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 lead to the problems described by Abel (2015a,c).

Example 5. Let $\texttt{D} : \texttt{Bool} \to \texttt{Set}$ be an indexed data type with two constructors $\texttt{tt} : \texttt{D}\ \texttt{true}$ and $\texttt{ff} : \texttt{D}\ \texttt{false}$. Then the conflict rule between $\texttt{tt}$ and $\texttt{ff}$ gives us the following equivalence:

$$(e_1 : \texttt{true} \equiv_{\texttt{Bool}} \texttt{false})(e_2 : \texttt{tt} \equiv_{\texttt{D}\ e_1} \texttt{ff}) \simeq \bot \tag{17}$$

On the other hand, we cannot apply the conflict rule if the first equation is between the *types* $\texttt{D}\ \texttt{true}$ and $\texttt{D}\ \texttt{false}$:

$$(e_1 : \texttt{D}\ \texttt{true} \equiv_{\texttt{Set}} \texttt{D}\ \texttt{false})(e_2 : \texttt{tt} \equiv_{e_1} \texttt{ff}) \not\simeq \bot \tag{18}$$

Allowing the conflict rule to apply in this case would mean that we have a way to distinguish between $\texttt{D}\ \texttt{true}$ and $\texttt{D}\ \texttt{false}$, which means that the type constructor $\texttt{D}$ is injective. In particular, this would be incompatible with the univalence axiom: there is an equivalence between $\texttt{D}\ \texttt{true}$ and $\texttt{D}\ \texttt{false}$ under which $\texttt{tt}$ is identified with $\texttt{ff}$, so univalence allows us to prove that $\texttt{D}\ \texttt{true} \equiv_{\texttt{Set}} \texttt{D}\ \texttt{false}$. 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 lead to the problems described by Danielsson (2010) and Vezzosi (2015).

4. Unification rules

Now that we know what the input and the output of a unification algorithm should be and have seen some examples of unification rules in action, we can start thinking about unification rules in general (Section 4.1). We also show the basic unification rules for simple and indexed data types (Section 4.2 and 4.3 respectively). Most of the work of constructing these equivalences has already been done by Cockx et al. (2014), we provide the final piece of the puzzle (Section 4.4).

By applying unification rules until there are no more equations left, we construct the most general unifier. 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 a strategy for choosing which rule to try first. When we discuss our implementation of a new unification algorithm for Agda in Section 7, we discuss how to choose such a strategy.

4.1 Unification rules as equivalences

Since the end result of the unification process (the most general unifier) is an equivalence f between $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ and Γ', we also represent unification rules as equivalences:

Definition 5. A *positive unification rule* is an equivalence of the form $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$.

These unification rules can then be chained together by transitivity of $\simeq$ to produce the most general unifier f.

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

Definition 6. A *negative unification rule* is an equivalence of the form $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$.

Before we go on to construct the basic unification rules, we give two easy but very useful manipulations on equivalences (and hence on unification rules): weakening and reordering. These two principles are used when we want to apply a unification rule, but the problem contains some additional flexible variables or equations that aren't mentioned in the rule. Note that we have already been using these lemmas in the examples in Section 3, for example in Example 1 to lift the solution rule over the variable $m : \mathbb{N}$.

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

Lemma 2. *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 can construct an equivalence $f : \Gamma \simeq \Gamma'$.*

Proof. The construction of the equivalence is in both cases very obvious, relying on $\texttt{J}$ to prove that the functions are mutual inverses. □

4.2 The basic unification rules

Now we are ready to give (typed versions of) the basic unification rules given by (McBride 1998b). The first two rules are generic in the sense that they work for any type A, while the three rules after that are specific to data types.

Solution. The solution rule allows us to solve an equation if one side is a variable. For any type A and term $t : A$, it is given by the equivalence

$$\texttt{solution} : (x : A)(e : x \equiv_A t) \simeq () \tag{19}$$

Note that the variable x should not occur freely in t. This rule is sometimes also called the *substitution rule* or the *coalescence rule* (in case t is also a variable).

Deletion. The deletion rule allows us to remove equations where the left- and right-hand side are definitionally equal. For any type A that satisfies $\texttt{K}$ and any term $t : A$, it is given by the equivalence

$$\texttt{deletion} : (e : t \equiv_A t) \simeq () \tag{20}$$

Note that this rule should not be used for types A that don't satisfy the $\texttt{K}$ rule (Cockx et al. 2014).

Injectivity. The injectivity rule allows us to simplify equations of the form $\texttt{c}\ x_1\ \ldots\ x_n = \texttt{c}\ y_1\ \ldots\ y_n$ if $\texttt{c}$ is a constructor. For example, it allows us to simplify the equation $\texttt{suc}\ m = \texttt{suc}\ n$ to $m = n$. Let $\texttt{c} : \Delta_{\texttt{c}} \to \texttt{D}$ be a constructor of the data type $\texttt{D} : \texttt{Set}_i$. The injectivity rule for $\texttt{c}$ is given by the equivalence

$$\texttt{injectivity}_{\texttt{c}} : (\texttt{c}\ \bar{x} \equiv_{\texttt{D}} \texttt{c}\ \bar{x}') \simeq (\bar{x} \equiv_{\Delta_{\texttt{c}}} \bar{x}') \tag{21}$$

where $\bar{x}, \bar{x}' : \Delta_{\texttt{c}}$. Note that the type of the equation on the left should be exactly $\texttt{D}$.

$$\frac{}{t_i \prec \mathtt{c}\ t_1\ \ldots\ t_n} \qquad \frac{f \prec t}{f\ s \prec t} \qquad \frac{r \prec s \qquad s \prec t}{r \prec t}$$

Figure 2. The structural order $s \prec t$ is used to check termination (Goguen et al. 2006) and to detect cyclical equations.

Conflict. The conflict rule allows us to detect absurd equations of the form $\mathtt{c}_1\ x_1\ \ldots\ x_m = \mathtt{c}_2\ y_1\ \ldots\ y_n$ 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 $\mathtt{zero} = \mathtt{suc}\ y$ is absurd. Let $\mathtt{c}_1 : \Delta_1 \to \mathtt{D}$ and $\mathtt{c}_2 : \Delta_2 \to \mathtt{D}$ be two distinct constructors of the data type $\mathtt{D} : \mathtt{Set}_i$. The conflict rule between $\mathtt{c}_1$ and $\mathtt{c}_2$ is given by the equivalence

$$\mathtt{conflict}_{\mathtt{c}_1,\mathtt{c}_2} : (\mathtt{c}_1\ \bar{x}_1 \equiv_{\mathtt{D}} \mathtt{c}_2\ \bar{x}_2) \simeq \bot \tag{22}$$

where $\bar{x}_1 : \Delta_1$ and $\bar{x}_2 : \Delta_2$. Again, the type of the equation should be exactly $\mathtt{D}$, see Example 2 of what can go wrong otherwise.

Cycle. The cycle rule allows us to detect cyclical equations of the form $x = \mathtt{c}\ y_1\ \ldots\ y_n$ where x occurs somewhere in $y_1, \ldots, y_n$. For example, it allows us to detect that an equation of the form $n = \mathtt{suc}\ n$ is absurd. To formalize this rule, we need the structural order on terms defined in Figure 2.[4] We say that x occurs *strongly rigid* in t if $x \prec t$.

Let $\mathtt{D} : \mathtt{Set}_i$ be a data type and let $x, t : \mathtt{D}$ be such that $x \prec t$. The cycle rule for x and t is given by the equivalence

$$\mathtt{cycle}_{x,t} : (x \equiv_{\mathtt{D}} t) \simeq \bot \tag{23}$$

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

4.3 Unification rules for indexed data types

The injectivity, conflict, and cycle rules defined in the previous section all work on regular data types. But unification only becomes really interesting once we consider indexed families of data types (Dybjer 1991). An indexed family $\mathtt{D} : \Xi \to \mathtt{Set}_i$ is defined by a telescope Ξ of indices and a list of constructors $\mathtt{c}_i : \Delta_i \to \mathtt{D}\ \bar{u}_i$ for $i = 1, \ldots, k$. Where the unification rules that we have seen so far only have a single equation on the left side, the rules for indexed data types have a telescope of equations: one equation for each index, and one final equation for the data type itself (see Examples 3 and 4).

Injectivity (indexed version). Let $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$ be a constructor of the data type $\mathtt{D} : \Xi \to \mathtt{Set}_i$. The injectivity rule for $\mathtt{c}$ is given by the equivalence

$$\begin{aligned}&\mathtt{injectivity}_{\mathtt{c}} :\\ &\quad (\bar{u}[\Delta \mapsto \bar{x}]; \mathtt{c}\ \bar{x} \equiv_{\overline{\mathtt{D}}} \bar{u}[\Delta \mapsto \bar{x}']; \mathtt{c}\ \bar{x}') \simeq (\bar{x} \equiv_{\Delta} \bar{x}')\end{aligned} \tag{24}$$

where $\bar{x}, \bar{x}' : \Delta$ and $\overline{\mathtt{D}}$ stands for the telescope $(\bar{u} : \Xi)(x : \mathtt{D}\ \bar{u})$.

Conflict (indexed 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 data type $\mathtt{D} : \Xi \to \mathtt{Set}_i$. The conflict rule between $\mathtt{c}_1$ and $\mathtt{c}_2$ is given by the equivalence

$$\begin{aligned}&\mathtt{conflict}_{\mathtt{c}_1,\mathtt{c}_2} :\\ &\quad (\bar{u}_1[\Delta_1 \mapsto \bar{x}_1]; \mathtt{c}_1\ \bar{x}_1 \equiv_{\overline{\mathtt{D}}} \bar{u}_2[\Delta_2 \mapsto \bar{x}_2]; \mathtt{c}_2\ \bar{x}_2) \simeq \bot\end{aligned} \tag{25}$$

where $\bar{x}_1 : \Delta_1$ and $\bar{x}_2 : \Delta_2$.

Cycle (indexed version). Let $\mathtt{D} : \Xi \to \mathtt{Set}_i$ be a data type and let $(\bar{u}; x), (\bar{v}; t) : \overline{\mathtt{D}}$ be such that $x \prec t$. The cycle rule is given by the equivalence

$$\mathtt{cycle}_{x,t} : (\bar{u}; x \equiv_{\overline{\mathtt{D}}} \bar{v}; t) \simeq \bot \tag{26}$$

[4] It is possible to define $x \prec t$ as a type using a technique similar to the one used by McBride et al. (2006) to define $\mathtt{Below}_{\mathtt{D}}$. However, this is not necessary as this type doesn't occur in the construction of $\mathtt{cycle}$, only the type $x \not\prec y$ as defined by McBride et al. (2006).

Note that for these three rules, the telescope for the equations on the left-hand side should be exactly $\overline{\mathtt{D}} = (\bar{u} : \Xi)(x : \mathtt{D}\ \bar{u})$. This is very convenient when the equations we start with are of this form because it allows us to simplify all equations at the same time. But what if the equations are not of this form? For example, we may have a single equation $\mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys$ without an accompanying equation between the indices. The example with the $\mathtt{Singleton}$ type from the introduction shows that it is not sound in general to just apply unification rules if the types don't correspond to the form given by the unification rule, but the above rules are actually overly restrictive. For the conflict and cycle rules at least, it is possible to generalize them to the case with arbitrary indices:

Lemma 3 (Generalized conflict)**.** *Let $\mathtt{D} : \Xi \to \mathtt{Set}_i$ be a data type and $\mathtt{c}_1 : \Delta_1 \to \mathtt{D}\ \bar{u}_1$ and $\mathtt{c}_2 : \Delta_2 \to \mathtt{D}\ \bar{u}_2$ two distinct constructors of $\mathtt{D}$. Let Φ furthermore be an arbitrary telescope with $\Phi \vdash \bar{v} : \Xi$, $\bar{s}_1, \bar{s}_2 : \Phi$, $\bar{t}_1 : \Delta_1$, and $\bar{t}_2 : \Delta_2$ such that $\bar{v}[\Phi \mapsto \bar{s}_1] = \bar{u}_1[\Delta_1 \mapsto \bar{t}_1]$ and $\bar{v}[\Phi \mapsto \bar{s}_2] = \bar{u}_2[\Delta_2 \mapsto \bar{t}_2]$ (both times a definitional equality).[5] Then we have*

$$\left(\bar{s}_1; \mathtt{c}_1\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2; \mathtt{c}_2\ \bar{t}_2\right) \simeq \bot \tag{27}$$

Note that while the indices $\bar{v}$ can be arbitrary (i.e. they don't have to be variables like in the standard conflict rule), the type of the final equation still has to be the data type $\mathtt{D}$ applied to indices, in particular it cannot be a variable itself (otherwise we run into the problem described in Example 5).

Before we start the actual proof, we first want to show how the naive proof attempt fails. It goes as follows: to construct a function $\left(\bar{s}_1; \mathtt{c}_1\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2; \mathtt{c}_2\ \bar{t}_2\right) \to \bot$, it suffices (by the $\mathtt{J}$ 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 can be constructed easily by calling the indexed conflict rule (25) 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 $\mathtt{J}$ 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 the error in this proof attempt stems from a confusion between the status of $\bar{s}_1$ and $\bar{s}_2$ as variables at the meta-level, but they can be arbitrary terms at the object level!

We work around this issue by using the following easy lemma:

Lemma 4. *For any $\bar{x}_1, \bar{x}_2 : \Phi$, $\bar{p} : \bar{x}_1 \equiv_{\Phi} \bar{x}_2$, and $y : \mathtt{D}\ \bar{v}[\Phi \mapsto \bar{x}_1]$, we have*

$$\begin{aligned}&\overline{\mathtt{subst}}\ (\lambda\bar{x}.\, \mathtt{D}\ \bar{v}[\Phi \mapsto \bar{x}])\ \bar{p}\ y \equiv_{\mathtt{D}\ \bar{v}[\Phi\mapsto\bar{x}_2]}\\ &\mathtt{subst}\ \mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\bar{x}.\, \bar{v}[\Phi \mapsto \bar{x}])\ \bar{p})\ y\end{aligned} \tag{28}$$

Proof. In contrast to the failed proof above, we can use $\mathtt{J}$ on $\bar{p}$ since there are no additional constraints on the structure of $\bar{x}_1$ and $\bar{x}_2$. After using $\mathtt{J}$, the remaining equality becomes $y \equiv_{\mathtt{D}\ \bar{v}[\Phi\mapsto\bar{x}_2]} y$, which can be proven by $\mathtt{refl}$. □

Proof (of Lemma 3). We start by expanding the definition of telescopic equality: we have to derive $\bot$ from

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_{\Phi} \bar{s}_2)\\ &(e_2 : \overline{\mathtt{subst}}\ (\lambda\bar{x}.\, \mathtt{D}\ \bar{v}[\Phi \mapsto \bar{x}])\ \bar{e}_1\ (\mathtt{c}_1\ \bar{t}_1)\\ &\qquad \equiv_{\mathtt{D}\ \bar{u}_2[\Delta_2\mapsto\bar{t}_2]} (\mathtt{c}_2\ \bar{t}_2))\end{aligned} \tag{29}$$

Substituting by the equality from Lemma 4 gives us that e_2 has type

$$\begin{aligned}&\mathtt{subst}\ \mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\bar{x}.\, \bar{v}[\Phi \mapsto \bar{x}])\ \bar{e}_1)\ (\mathtt{c}_1\ \bar{t}_1)\\ &\quad \equiv_{\mathtt{D}\ \bar{u}_2[\Delta_2\mapsto\bar{t}_2]} (\mathtt{c}_2\ \bar{t}_2)\end{aligned} \tag{30}$$

[5] Note that these definitional equalities are always satisfied when the telescope in (27) is well-typed, and vice versa.

So now we can call the conflict rule (25) with the arguments $\texttt{cong}\ (\lambda\bar{x}.\,\bar{v}[\Phi \mapsto \bar{x}])\ \bar{e}_1; e_2$ 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 5 (Generalized cycle). *Let* $\mathtt{D} : \Xi \to \mathtt{Set}_i$ *be a data type and let* Φ *be an arbitrary telescope with* $\Phi \vdash \bar{v} : \Xi$, $\bar{s}_1, \bar{s}_2 : \Phi$, $t_1 : \mathtt{D}\ \bar{v}[\Phi \mapsto \bar{s}_1]$, *and* $t_2 : \mathtt{D}\ \bar{v}[\Phi \mapsto \bar{s}_2]$ *such that* $t_1 \prec t_2$. *Then we have*

$$\left(\bar{s}_1; t_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2; t_2\right) \simeq \bot \tag{31}$$

Proof. Analogously to the proof of Lemma 3. □

For injectivity, it isn't possible to generalize the unification rule to arbitrary indices like we just did for conflict and cycle. The reason is that 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 Section 6.2, we describe a general method for generalizing the indices of $\mathtt{D}$ until they are in the correct form to apply the injectivity rule. However, in the special (but common) case where the index telescope Ξ satisfies the K rule, it is possible to construct the generalization, thus avoiding the need for the additional machinery in Section 6.2:

Lemma 6 (Generalized injectivity). *Let* $\mathtt{D} : \Xi \to \mathtt{Set}_i$ *be a data type with (at least) one constructor* $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$, *and assume* Ξ *satisfies* K, *i.e. we have* $\mathtt{deletion}_\Xi : (\bar{e} : \bar{x} \equiv_\Xi \bar{x}) \simeq ()$ *for all* $\bar{x} : \Xi$. *Let* Φ *furthermore be an arbitrary telescope with* $\Phi \vdash \bar{v} : \Xi$, $\bar{s}_1, \bar{s}_2 : \Phi$, $\bar{t}_1, \bar{t}_2 : \Delta$ *such that* $\bar{v}[\Phi \mapsto \bar{s}_1] = \bar{u}_1[\Delta \mapsto \bar{t}_1]$ *and* $\bar{v}[\Phi \mapsto \bar{s}_2] = \bar{u}_2[\Delta \mapsto \bar{t}_2]$ *(both times a definitional equality). Then we have*

$$\left(\bar{s}_1; \mathtt{c}\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2; \mathtt{c}\ \bar{t}_2\right) \simeq (\bar{s}_1; \bar{t}_1 \equiv_{\Phi\Delta} \bar{s}_2; \bar{t}_2) \tag{32}$$

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

Proof. As for the previous lemma, we expand the definition of telescopic equality and apply Lemma 4 to get to

$$\begin{array}{l}(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)\\ (e_2 : \mathtt{subst}\ \mathtt{D}\ (\mathtt{cong}\ (\lambda\bar{x}.\,\bar{v}[\Phi \mapsto \bar{x}])\ \bar{e}_1)\ (\mathtt{c}\ \bar{t}_1) \equiv (\mathtt{c}\ \bar{t}_2))\end{array} \tag{33}$$

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

$$\begin{array}{l}(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)\\ (\bar{e}'_1 : \bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2])\\ (e_2 : \mathtt{subst}\ \mathtt{D}\ (\mathtt{cong}\ (\lambda\bar{x}.\,\bar{v}[\Phi \mapsto \bar{x}])\ \bar{e}_1)\ (\mathtt{c}\ \bar{t}_1) \equiv (\mathtt{c}\ \bar{t}_2))\end{array} \tag{34}$$

Again by K, we have that the proofs $\mathtt{cong}\ (\lambda\bar{x}.\,\bar{v}[\Phi \mapsto \bar{x}])\ \bar{e}_1$ and $\bar{e}'_1$ of type $\bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2]$ are equal. Substituting the latter for the former yields the telescope

$$\begin{array}{l}(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)\\ (\bar{e}'_1 : \bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2])\\ (e_2 : \mathtt{subst}\ \mathtt{D}\ \bar{e}'_1\ (\mathtt{c}\ \bar{t}_1) \equiv (\mathtt{c}\ \bar{t}_2))\end{array} \tag{35}$$

Finally, we can apply the injectivity rule (24) 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 whole telescope is equivalent to

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}_2 : \bar{t}_1 \equiv_\Delta \bar{t}_2) \tag{36}$$

which is what we wanted to prove. □

4.4 Construction of the unification rules

Construction of $\mathtt{solution}$ **(19).** Note that the construction of the functions $\mathtt{solution} : (x : A)(x \equiv_A t) \to ()$ and $\mathtt{isRinv}\ \mathtt{solution} : () \to () \equiv_{()} ()$ is trivial since they both target an empty telescope. The function $\mathtt{solution}^{-1} : () \to (x : A)(e : x \equiv_A t)$ is defined by $\mathtt{solution}^{-1}\ () = t; \mathtt{refl}$, and $\mathtt{isLinv}\ \mathtt{solution} : (x : A)(e : x \equiv_A t) \to t; \mathtt{refl} \equiv_{(x:A)(e:x\equiv_A t)} x; e$ is a direct application of the J rule.

Construction of $\mathtt{deletion}$ **(20).** The construction is similar to the construction of $\mathtt{solution}$, except that we use K instead of J.

Construction of $\mathtt{injectivity_c}$ **(24).** The injectivity rule is an instance of the more general principle of "no confusion" (McBride et al. 2006) where the left- and right-hand sides have been instantiated to applications of the same constructor $\mathtt{c}$. Cockx et al. (2014) gave a left inverse $\mathtt{noConf}^{-1}$ to the function $\mathtt{noConf}$, so all that's left to do is to prove that it is a right inverse as well.

We do not repeat the full construction of $\mathtt{noConf}$ and $\mathtt{noConf}^{-1}$ here, but merely recall their essential computational properties. First, the type $\mathtt{NoConfusion_D} : \bar{\mathtt{D}} \to \bar{\mathtt{D}} \to \mathtt{Set}_d$ is constructed such that

$$\begin{array}{ll}\mathtt{NoConfusion_D}\ (\bar{u}; \mathtt{c}\ \bar{s})\ (\bar{v}; \mathtt{c}\ \bar{t}) = \bar{s} \equiv_{\Delta_\mathtt{c}} \bar{t} & \\ \mathtt{NoConfusion_D}\ (\bar{u}; \mathtt{c}\ \bar{s})\ (\bar{v}; \mathtt{c}'\ \bar{t}) = \bot & (\text{when } \mathtt{c} \neq \mathtt{c}')\end{array} \tag{37}$$

Next, $\mathtt{noConf_D} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \bar{x} \equiv_{\bar{\mathtt{D}}} \bar{y} \to \mathtt{NoConfusion_D}\ \bar{x}\ \bar{y}$ and $\mathtt{noConf^{-1}}_\mathtt{D} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \mathtt{NoConfusion_D}\ \bar{x}\ \bar{y} \to \bar{x} \equiv_{\bar{\mathtt{D}}} \bar{y}$ are defined such that

$$\begin{array}{r}\mathtt{noConf}\ (\bar{u}; \mathtt{c}\ \bar{s})\ (\bar{u}; \mathtt{c}\ \bar{s})\ \mathtt{refl} = \mathtt{refl}\\ \mathtt{noConf}^{-1}\ (\bar{u}; \mathtt{c}\ \bar{s})\ (\bar{u}; \mathtt{c}\ \bar{s})\ \mathtt{refl} = \mathtt{refl}\end{array} \tag{38}$$

for all constructors $\mathtt{c}$ of $\mathtt{D}$. To construct the proof $\mathtt{isRinv}$ that

$$\begin{array}{l}(\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to (e : \mathtt{NoConfusion_D}\ \bar{x}\ \bar{y})\\ \quad \to \mathtt{noConf}\ \bar{x}\ \bar{y}\ (\mathtt{noConf}^{-1}\ \bar{x}\ \bar{y}\ e) \equiv e\end{array} \tag{39}$$

we first apply case analysis on $\bar{x}$ and $\bar{y}$. In the cases where we have two distinct constructors $\mathtt{c}$ and $\mathtt{c}'$, we have $e : \bot$ so we can conclude by $\mathtt{elim}_\bot$. In the diagonal cases we have $e : \bar{s} \equiv_{\Delta_\mathtt{c}} \bar{t}$. Eliminating these equations with J leaves us with the goal $\mathtt{refl} \equiv_{\bar{s}\equiv\bar{s}} \mathtt{refl}$, which we can solve by giving $\mathtt{refl}$.

Construction of $\mathtt{conflict_{c_1,c_2}}$ **(25).** As for injectivity, the construction of $\mathtt{conflict_{c_1,c_2}}$ is just a special case of $\mathtt{noConf_D}$. The only difference is that the target type is $\bot$, so it is automatically an equivalence.

Construction of $\mathtt{cycle}_{x,x'}$ **(26).** The construction of $\mathtt{cycle}_{x,x'}$ is also given by Cockx et al. (2014). As for the conflict rule, this function is automatically an equivalence because the target is $\bot$.

5. Computational behaviour

Until now, we have only been interested in an equivalence representing a most general unifier insofar that it has the correct type. But as a term in type theory, it also has a certain computational behaviour. This computational behaviour may be important for the applications we have in mind. In particular, 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 $m' : (\bar{x}' : \Gamma') \to T\ (f^{-1}\ \bar{x}')$ we expect m to have 'the same' computational behaviour as m' in case the equations $\bar{u} \equiv_\Delta \bar{v}$ are actually satisfied.

The computational behaviour of unifiers is also important for practical purposes when implementing dependent pattern matching: instead of introducing a new symbol m', the system may ask the user to define $m\ (f^{-1}\ \bar{x}')$. This is the idea behind *inaccessible patterns* (also known as *dot patterns*): the values of these inaccessible patterns are determined by f^{-1}. The fact that $m\ (f^{-1}\ \bar{x}')$ evaluates to $m'\ \bar{x}'$ ensures that the computational behaviour corresponds to the clause written by the user (Goguen et al. 2006).

To make this idea precise, note that if $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ is a most general unifier, the arguments of m for which $\bar{u}$ and $\bar{v}$ are equal are given exactly by the image of f^{-1}. So we need that $m\ (f^{-1}\ \bar{x}')$ evaluates to $m'\ \bar{x}'$ for any $\bar{x}' : \Gamma'$. By the definition (9) of m, $m\ (f^{-1}\ \bar{x}')$ is equal to

$$\overline{\mathtt{subst}}\ T\ (\mathtt{isLinv}\ f\ (f^{-1}\ \bar{x}'))\ (m'\ (f\ (f^{-1}\ \bar{x}'))) \tag{40}$$

This evaluates to $m'\ \bar{x}'$ under the following conditions:

Definition 7. An equivalence $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ is a *strong unifier* if for any $\bar{x}' : \Gamma'$, it satisfies the following two computational properties:

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

Note that $\mathtt{isLinv}\ f\ (f^{-1}\ \bar{x}')$ has type $f^{-1}\ (f\ (f^{-1}\ \bar{x}')) \equiv f^{-1}\ \bar{x}'$, which reduces to $f^{-1}\ \bar{x}' \equiv f^{-1}\ \bar{x}'$ because of the first property. So we already know that the type of the equation is reflexive, the second requirement is only meant to ensure that the proof $\mathtt{isLinv}\ f\ (f^{-1}\ \bar{x}')$ itself evaluates to $\overline{\mathtt{refl}}$ as well.

Note also that even if we had only established so far that f^{-1} is a left inverse to f, this first property gives us a very easy way to prove that it is also a right inverse: we can simply define $\mathtt{isRinv}\ f : (\bar{x}' : \Delta') \to f\ (f^{-1}\ \bar{x}') \equiv_{\Delta'} \bar{x}'$ by $\mathtt{isRinv}\ f\ \bar{x}' = \overline{\mathtt{refl}}$.

At first sight, it seems natural to require that each unification rule satisfies the same conditions on their computational behaviour as those for a strong unifier (see Definition 7). However, it turns out that this requirement is too strong: for example, the rule $\mathtt{injectivity_{suc}} : (\mathtt{suc}\ m \equiv_\mathbb{N} \mathtt{suc}\ n) \simeq (m \equiv_\mathbb{N} n)$ doesn't satisfy $\mathtt{injectivity_{suc}}\ (\mathtt{injectivity_{suc}}^{-1}\ e) = e$ for arbitrary $e : m \equiv_\mathbb{N} n$, since $\mathtt{injectivity_{suc}}^{-1}\ e$ only reduces once the proof e is $\mathtt{refl}$. Instead, we give a different criterion that implies the criterion in Definition 7 in case the telescope of equations on the right is empty:

Definition 8. 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 $\bar{t}' : \Gamma'$ such that $\bar{u}'[\Gamma' \mapsto \bar{t}'] = \bar{v}'[\Gamma' \mapsto \bar{t}']$ (definitionally), it satisfies the following three computational properties:

- $r^{-1}\ \bar{t}'\ \overline{\mathtt{refl}}$ is of the form $\bar{u}; \overline{\mathtt{refl}}$ for some $\bar{u} : \Gamma$
- $r\ (r^{-1}\ \bar{t}'\ \overline{\mathtt{refl}}) = \bar{t}'; \overline{\mathtt{refl}}$
- $\mathtt{isLinv}\ r\ (r^{-1}\ \bar{t}'\ \overline{\mathtt{refl}}) = \overline{\mathtt{refl}}$

Lemma 7. *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. We verify the two properties of a strong unifier:

- Since each of the telescope mappings $r_1^{-1}, \ldots, r_n^{-1}$ satisfy the first property of a strong unification rule, we know that the component of $r_n^{-1}\ \ldots (r_2^{-1}\ (r_1^{-1}\ \bar{x}'\ ()))$ that has type $\bar{u} \equiv_\Delta \bar{v}$ evaluates to $\overline{\mathtt{refl}}$. Hence we can compute:

$$\begin{aligned} f\ (f^{-1}\ \bar{x}') &= r_1\ \ldots (r_n\ (r_n^{-1}\ \ldots (r_1^{-1}\ \bar{x}'\ ()))) \\ &= r_1\ \ldots (r_{n-1}\ (r_{n-1}^{-1}\ \ldots (r_1^{-1}\ \bar{x}'\ ()))) \\ &= \ldots = r_1\ (r_1^{-1}\ \bar{x}'\ \overline{\mathtt{refl}}) = \bar{x}' \end{aligned} \tag{41}$$

- To verify the second property of a strong unifier, note that $\mathtt{isLinv}\ f\ \bar{x}\ \bar{e}$ is constructed by composing the individual proofs $\mathtt{isLinv}\ r_i\ (r_{i+1}\ \ldots (r_n\ \bar{x}\ \bar{e}))$ by transitivity. If we fill in $f^{-1}\ \bar{x}' = r_n^{-1}\ \ldots\ (r_1^{-1}\ \bar{x}'\ ())$ for $\bar{x}; \bar{e}$, then this evaluates (by the first and second properties of a strong unification rule) to $\mathtt{isLinv}\ r_i\ (r_i^{-1}\ (r_{i-1}^{-1}\ \ldots\ (r_1^{-1}\ \bar{x}'\ ())))$. Now we can apply the third property (again together with the first) to see that this evaluates to $\overline{\mathtt{refl}}$ indeed. Since $\overline{\mathtt{refl}}$ is the identity for composition, we have $\mathtt{isLinv}\ f\ (f^{-1}\ \bar{x}') = \overline{\mathtt{refl}}$ as we wanted to prove.

□

Lemma 8. *The solution* (19)*, deletion* (20)*, and injectivity* (24) *rules are strong unification rules.*

Proof. This follows directly from the construction of these rules (see Section 4.4). □

6. More unification rules

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. In this section, we show three examples of unification rules that go beyond the basic unification rules used by Goguen et al. (2006) and Cockx et al. (2014): eta rules for record types (Section 6.1), reverse unification rules (Section 6.2), and an example of a unification rule dealing with *higher inductive types* (Section 6.3).

The development of these rules is at the moment of writing still in various stages of progress: the eta rules are finished and implemented as part of our new unifier for Agda (see Section 7), the reverse unification rules are also finished for simple data types but not yet implemented, and the rules for higher inductive type is only one example with no general theory so far.

6.1 Eta rules for record types

We start with two simple but very useful unification rules that deal with η-equality of records. A *record type* is a type for grouping values together. A record type $\mathtt{R} : \mathtt{Set}_i$ is defined by a number of fields (or projections)

$$\begin{aligned} \mathtt{f_1} &: (r : R) \to A_1 \\ \mathtt{f_2} &: (r : R) \to A_2\ (\mathtt{f_1}\ r) \\ &\vdots \\ \mathtt{f_n} &: (r : R) \to A_n\ (\mathtt{f_1}\ r)\ \ldots\ (\mathtt{f_{n-1}}\ r) \end{aligned} \tag{42}$$

Note that 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 $\mathtt{snd} : (p : \Sigma_{x:A}\ (B\ x)) \to B\ (\mathtt{fst}\ p)$.

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

$$\mathtt{f_i}\ (\mathtt{record}\{\mathtt{f_1} = x_1; \ldots; \mathtt{f_n} = x_n\}) = x_i \tag{43}$$

One of the properties that sets a record type apart from a regular data type 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). The η-law for the record $\mathtt{R}$ states that for any $r : R$, we have

$$r = \mathtt{record}\{\mathtt{f_1} = \mathtt{f_1}\ r; \ldots; \mathtt{f_n} = \mathtt{f_n}\ r\} \tag{44}$$

We use this 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. [6]

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

η-expansion of a variable. Let $\mathtt{R} : \mathtt{Set}_i$ be the record type with fields given by (42). Then the rule $\eta\mathtt{var}_{\mathtt{R}}$ is the equivalence:

$$\eta\mathtt{var}_{\mathtt{R}} : (r : \mathtt{R}) \simeq (f_1 : A_1)\dots(f_n : A_n\ f_1\ \dots\ f_{n-1}) \qquad (45)$$

Example 6. This rule is especially useful for solving equations where one side is a projection applied to a variable, for example:

$$\begin{aligned}&(\underline{p} : \mathbb{N} \times \mathbb{N})(e : \mathtt{fst}\ p \equiv_{\mathbb{N}} \mathtt{zero})\\ \simeq\ &(\underline{x} : \mathbb{N})(y : \mathbb{N})(\underline{e} : x \equiv_{\mathbb{N}} \mathtt{zero})\\ \simeq\ &(y : \mathbb{N})\end{aligned} \qquad (46)$$

where $A \times B$ stands for $\Sigma_{(x:A)}(\lambda x.\,B)$ if B doesn't depend on x.

The construction of $\eta\mathtt{var}_{\mathtt{R}}$ is straightforward: $\eta\mathtt{var}\ \ r$ is defined by $\mathtt{f_1}\ r;\dots;\mathtt{f_n}\ r$, while $\eta\mathtt{var}^{-1}\ f_1\ \dots\ f_n$ is defined by $\mathtt{record}\{\mathtt{f_1} = f_1;\dots;\mathtt{f_n} = f_n\}$. The proofs of both $\mathtt{isLinv}$ and $\mathtt{isRinv}$ are simply $\mathtt{refl}$: in the former case this is type-correct because of the η-law (44), and in the latter case because of the computation rules for projections (43).

The $\eta\mathtt{var}$ rule also satisfies all the computational properties of a strong unification rule. The first property is trivial as this rule does not involve equations. The second property also holds since $\mathtt{record}\{\mathtt{f_1} = \mathtt{f_1}\ r;\dots;\mathtt{f_n} = \mathtt{f_n}\ r\} = r$ by the η law. Finally, the third property holds as well since $\mathtt{isLinv}\ \ \eta\mathtt{var}\ \ r$ is *always* definitionally equal to $\mathtt{refl}$.

η-expansion of an equation. Let $\mathtt{R} : \mathtt{Set}_i$ again be a record type with fields given by (42). Then $\eta\mathtt{eq}_{\mathtt{R}}$ is the equivalence:

$$\eta\mathtt{eq} : (e : r \equiv_R s) \simeq \begin{array}{l}(e_1 : \mathtt{f_1}\ r \equiv_{A_1} \mathtt{f_1}\ s)\dots\\ (e_n : \mathtt{f_n}\ r \equiv_{A_n\ e_1\ \dots\ e_{n-1}} \mathtt{f_n}\ s)\end{array} \qquad (47)$$

Example 7. This rule is useful when one side of an equation is of the form $\mathtt{record}\{\dots\}$. For example, if $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, then

$$\begin{aligned}&(x\ y\ z : \mathbb{N})(\underline{e} : x, y \equiv_{\mathbb{N}\times\mathbb{N}} f\ z)\\ \simeq\ &(\underline{x}\ y\ z : \mathbb{N})(\underline{e_1} : x \equiv_{\mathbb{N}} \mathtt{fst}\ (f\ z))(e_2 : y \equiv_{\mathbb{N}} \mathtt{snd}\ (f\ z))\\ \simeq\ &(\underline{y}\ z : \mathbb{N})(\underline{e_2} : y \equiv_{\mathbb{N}} \mathtt{snd}\ (f\ z))\\ \simeq\ &(z : \mathbb{N})\end{aligned} \qquad (48)$$

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 $\eta\mathtt{eq}^{-1}\ \bar{e} = \mathtt{cong}\ \ \eta\mathtt{var}^{-1}\ \bar{e}$. The proofs of $\mathtt{isLinv}$ and $\mathtt{isRinv}$ are straightforward applications of J. The computational behaviour of $\eta\mathtt{eq}$ is also trivially correct since $\eta\mathtt{eq}$, $\eta\mathtt{eq}^{-1}$, $\mathtt{isLinv}\ \eta\mathtt{eq}$ and $\mathtt{isRinv}\ \eta\mathtt{eq}$ all map $\mathtt{refl}$ to $\mathtt{refl}$.

6.2 Reverse unification rules

In Section 4.3 we noted that the unification rules for indexed data types require that the type of the final equation is the data type applied to equation variables. We also saw that this restriction can be loosened for the $\mathtt{conflict}$ and $\mathtt{cycle}$ rules (Lemma 3 and 5), and also for the $\mathtt{injectivity}$ rule in case the index types satisfy K (Lemma 6). But there are plenty of cases where we want to apply injectivity but the type is not of this form.

Luckily, we can still make progress by first generalizing the indices in the type of the equation before we apply injectivity. We do this by applying the solution and injectivity rules in the reverse direction. This is possible since equivalence is a symmetric relation. However, it is necessary to check that the reversed rule still has good computational behaviour as described in Definition 8.

Example 8. For example, let $\mathtt{Singleton} : A \to \mathtt{Set}$ be the data type from the introduction (with one constructor $\mathtt{sing} : (x : A) \to \mathtt{Singleton}\ \ A\ \ x$) and consider the unification problem $\mathtt{sing}\ \ x\ \ \equiv_{\mathtt{Singleton}\ x}\ \mathtt{sing}\ \ x$. Then we cannot apply $\mathtt{injectivity}_{\mathtt{sing}}$ since the index x is a regular variable rather than an equation variable. To transform the unification problem to the required form, we can apply the $\mathtt{solution}$ rule (19) in the reverse direction. Instead of solving an equation, this introduces a new variable y and an equation $e_1 : x \equiv_A y$:

$$\begin{aligned}&(\underline{x} : A)(e : \mathtt{sing}\ x \equiv_{\mathtt{Singleton}\ x} \mathtt{sing}\ x)\\ \simeq\ &(x\ y : A)(e_1 : x \equiv_A y)(\underline{e_2} : \mathtt{sing}\ x \equiv_{\mathtt{Singleton}\ e_1} \mathtt{sing}\ y)\\ \simeq\ &(x\ \underline{y} : A)(\underline{e} : x \equiv_A y)\\ \simeq\ &(x : A)\end{aligned} \qquad (49)$$

Example 9. Sometimes, it is necessary to apply $\mathtt{injectivity}$ (21) in the reverse direction as well as $\mathtt{solution}$. For example, if we have an equation $\mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys$ then we cannot apply $\mathtt{injectivity}_{\mathtt{cons}}$ straight away. Instead, we first apply the rules $\mathtt{solution}$ and $\mathtt{injectivity}_{\mathtt{suc}}$ in the reverse direction to transform the type of this equation:

$$\begin{array}{ll}&(\underline{n} : \mathbb{N})(x\ y : A)(xs\ ys : \mathtt{Vec}\ A\ n)\\ &(e : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys)\\ \simeq & \begin{array}{l}(m\ n : \mathbb{N})(x\ y : A)(xs : \mathtt{Vec}\ A\ m)(ys : \mathtt{Vec}\ A\ n)\\ (\underline{e_1} : m \equiv_{\mathbb{N}} n)\\ (e_2 : \mathtt{cons}\ m\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ e_1)} \mathtt{cons}\ n\ y\ ys)\end{array}\\ \simeq & \begin{array}{l}(m\ n : \mathbb{N})(x\ y : A)(xs : \mathtt{Vec}\ A\ m)(ys : \mathtt{Vec}\ A\ n)\\ (e_1 : \mathtt{suc}\ m \equiv_{\mathbb{N}} \mathtt{suc}\ n)\\ (\underline{e_2} : \mathtt{cons}\ m\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys)\end{array}\\ \simeq & \begin{array}{l}(\underline{m}\ n : \mathbb{N})(x\ \underline{y} : A)(xs : \mathtt{Vec}\ A\ m)(\underline{ys} : \mathtt{Vec}\ A\ n)\\ (\underline{e_1} : m \equiv_{\mathbb{N}} n)(\underline{e_2} : x \equiv_A y)(\underline{e_3} : xs \equiv_{\mathtt{Vec}\ A\ e_1} ys)\end{array}\\ \simeq & (n : \mathbb{N})(x : A)(xs : \mathtt{Vec}\ A\ n)\end{array} \qquad (50)$$

Here is a general strategy for applying reverse unification rules:

- If an index is a regular variable $x : A$ instead of an equation variable, we 'duplicate' the variable by applying $\mathtt{solution}$ in reverse. This introduces an additional variable $y : A$ and an equation $e : x \equiv_A y$.
- If an index is a constructor $\mathtt{c}$ of a simple data type, first make sure that all the arguments of this constructor are equation variables (by applying this strategy recursively), and then apply $\mathtt{injectivity}_{\mathtt{c}}$ in reverse.
- If an index is not a variable or a constructor, or all indices are variables but they are not distinct, then we give up.

The injectivity rule for constructors of indexed data types (24) is much harder to apply in the reverse direction.

Example 10. Let $f : A \to B$ be a function and $\mathtt{IM}\ f : (x : A) \to \mathtt{Im}\ \ f\ \ (f\ \ x) \to \mathtt{Set}$ be a data type with one constructor $\mathtt{IMAGE} : (x : A) \to \mathtt{IM}\ A\ x\ (\mathtt{image}\ x)$ (This type characterizes the graph of the constructor $\mathtt{image}$). Then we can apply the $\mathtt{injectivity}_{\mathtt{image}}$ rule in reverse to solve the following unification problem:

$$\begin{array}{ll}&(x\ y : A)(e_1 : x \equiv_A y)\\ &(e_2 : \mathtt{IMAGE}\ x \equiv_{\mathtt{IM}\ f\ (f\ e_1)\ (\mathtt{image}\ e_1)} \mathtt{IMAGE}\ y)\\ \simeq & \begin{array}{l}(x\ y : A)(e_1 : f\ x \equiv_A f\ y)\\ (e_2 : \mathtt{image}\ x \equiv_{\mathtt{Im}\ f\ e_1} \mathtt{image}\ y)\\ (e_3 : \mathtt{IMAGE}\ x \equiv_{\mathtt{IM}\ f\ e_1\ e_2} \mathtt{IMAGE}\ y)\end{array}\\ \simeq & (x\ y : A)(e : x \equiv_A y)\\ \simeq & (x : A)\end{array} \qquad (51)$$

In general, it may be difficult to apply the injectivity rule for constructors of indexed data types automatically as we need to match the indices in the type of the equation against the indices in the type of the constructor.

6.3 Unification for higher inductive types

Higher inductive types are an interesting new concept from HoTT. They are defined like regular inductive types, except that they can also have *path constructors* that introduce additional equalities between elements of the type. This means that they do not necessarily satisfy the injectivity, conflict, and cycle rules anymore. But it is possible to construct new unification rules for higher inductive types on a case-by-case basis.

Example 11. The interval $\mathtt{I}$ is a higher inductive type with two point constructors $\mathtt{0} : \mathtt{I}$ and $\mathtt{1} : \mathtt{I}$ and one path constructor $\mathtt{line} : \mathtt{0} \equiv_{\mathtt{I}} \mathtt{1}$. We have the following equivalence:

$$\mathtt{contract} : (e : \mathtt{0} \equiv_{\mathtt{I}} \mathtt{1}) \simeq () \tag{52}$$

Note that $\mathtt{contract}^{-1}\ () = \mathtt{line}$, 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, any equivalence constructed by the encode-decode method (Licata and Shulman 2013; McKinna and Forsberg 2015) could in principle be used as a unification rule. 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 in order to deal with these variables, so we see this as an interesting direction for future work.

7. Implementation

As we mentioned in the introduction, our main motivation for studying unification in a dependently typed setting is for typechecking definitions by dependent pattern matching, and especially to get a workable and sound approach for dealing with heterogeneous equations. So how does an algorithm based on unifiers as equivalences work out in practice? We rewrote the unification algorithm used by the Agda language according to the ideas presented in this paper. This resulted in both a number of bugfixes (Section 7.1) and a much cleaner implementation than before (Section 7.2).

7.1 Impact on the Agda user

From the point of view of a user of Agda, unification is something that 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. That our implementation satisfies the latter can be seen from the fact that our implementation directly resulted in a fix for issue #1408 (Vezzosi 2015) on the Agda bug tracker, dealing with an incompatibility between heterogeneous equations and the –without-K option. Equally important, our implementation provides a much more principled solution to issues #292 (Danielsson 2010), #1071 (Danielsson 2014), #1406 (Abel 2015a), #1411 (Abel 2015b), and #1427 (Abel 2015c). 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 useful cases, see for example issue #1435 (Danielsson 2015).

```
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. The data types used for representing unification states and unification rules closely follow the theory. In addition to the unification rules presented in this paper, Agda also has unification rules for dealing with literals, sized types (Abel 2010) and irrelevant equations (Abel 2011), features not discussed in this paper. There is also a rule for injective type constructors that is only used when this is enabled explicitly by the user.

The addition of the new unification rules for η-equality of record values also improved the way Agda handles records a lot. In particular, the addition of these rules fixed the issues #635 (Peebles 2012) and #1613 (Abel 2015d), and provides a more principled solution to issue #473 (Danielsson 2011).

There is still a small number of cases where our implementation fails to apply the injectivity rule, since we haven't yet implemented the reverse unification rules described in Section 6.2. Luckily, these cases only occur when the –without-K option is enabled, for otherwise we can apply Lemma 6. Still, we would like to implement these reverse unification rules as well in the future.

7.2 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. The unification strategy takes a unification state as an argument and produces a list of unification rules to try (see Figure 4), while the unification engine tries to apply these rules one by one until one succeeds (see Figure 5).

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 in order to avoid heterogeneous equations as much as possible, and this was also the preferred

```
type UnifyStrategy =
  UnifyState -> ListT TCM UnifyStep

skipIrrelevantStrategy basicUnifyStrategy
dataStrategy literalStrategy etaExpandVarStrategy
etaExpandEquationStrategy injectiveTypeConStrategy
simplifySizesStrategy checkEqualityStrategy
  :: Int -> UnifyStrategy
```

Figure 4. 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 5. 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.

method for the old algorithm. However, our unification rules for indexed data types actually benefit from having unsolved equations in the telescope, so a unification algorithm that starts from the right has 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.

8. Related Work

Unification is a large area of research that we cannot hope to cover here. 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 discussed, such as higher-order unification and E-unification, are orthogonal to the work in this paper, although it would be interesting to see how they fit within our framework.

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 very analogous to the type-theoretic ones presented in this paper. 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 data types, 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 data types. McBride (2002) solves this by introducing a heterogeneous identity type. However, the K axiom 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. Cockx et al. (2014) solves the problem of requiring the K axiom, but the unification rules still only work on the first equation in a telescope. As a consequence, they have to limit the injectivity, conflict, and cycle rules to work only in homogeneous situations, while we can use them in their fully general form.

Our approach is closely related to the notion of inversion of an inductive hypothesis (Cornes and Terrasse 1995; Monin 2010). Roughly speaking, inversion works by crafting a special-purpose *diagonalizer* that is used as the motive for an eliminator. Alternatively, unification can be used as an different approach for proving inversion lemmas (McBride 1998b). The process of constructing this diagonalizer has recently also been automated (Braibant 2013). The advantage of the diagonalizer approach is that it moves most of the work to the type level, potentially improving performance of the resulting function. 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. it may be non-linear). It would be interesting to try to implement an inversion tactic based on the unification algorithm in this paper to see how the two approaches compare.

Type checkers of dependently typed languages typically have some facility for meta-variables that are solved by higher order pattern unification. This is not directly related to the work in this paper because the requirements on the unification algorithm are very different. 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, thereby giving up on finding most general unifiers in favour of finding solutions more often (Ziliani and Sozeau 2015). 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 an interesting line of research to further investigate the similarities and differences between these two unification algorithms.

9. Conclusion and future work

The main advantage of a dependently typed language over a simply typed one is the possibility to express and enforce correctness properties in the language itself, rather than externally. In this paper, we apply this idea to unification by demanding evidence of unification in the form of an equivalence between telescopic equalities. This concept of unifiers as equivalences allows us to give a precise correctness criterion for unification rules, and guides us in the implementation of a cleaner and more correct version of the unification engine used for dependent pattern matching in Agda.

In this work, we focus on one application, 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.

In the future, 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) corresponds to new unification rules for higher inductive types. 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?

Another interesting prospect is to put the power of unification in the hands of the user by allowing them to define custom unification rules. For example, if the user can provide a proof of $(f \enspace x \enspace \equiv_B$

$f\ y) \simeq (x \equiv_A y)$ for some function $f : A \to B$, then this could be used as an injectivity rule for f by the unifier.

Finally, there is the question if we can 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 7. This would definitely be a big step towards a verified typechecker for a dependently typed language implemented in the language itself.

References

A. Abel. MiniAgda: Integrating sized and dependent types. In *Workshop on Partiality and Recursion in Interactive Theorem Provers (PAR)*, 2010.

A. Abel. Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures*. 2011.

A. 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.

A. Abel. Order of patterns matters for checking left hand sides, 2015b. URL `https://github.com/agda/agda/issues/1411`. On the Agda bug tracker.

A. 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.

A. 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.

F. Baader and W. Snyder. Unification theory. *Handbook of automated reasoning*, 2001.

T. Braibant. A new Coq tactic for inversion, 2013. URL `http://gallium.inria.fr/blog/a-new-Coq-tactic-for-inversion`.

J. Cockx, D. Devriese, and F. Piessens. Pattern matching without K. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. ACM, 2014.

C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom, 2015. Preprint.

T. Coquand. Pattern matching with dependent types. In *Types for proofs and programs*, 1992.

C. Cornes and D. Terrasse. Automating inversion of inductive predicates in Coq. In *Types for Proofs and Programs*. 1995.

N. A. 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.

N. A. Danielsson. The unification machinery does not respect η-equality, 2011. URL `https://github.com/agda/agda/issues/473`. On the Agda bug tracker.

N. A. 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.

N. A. Danielsson. Dependent pattern matching is broken, 2015. URL `https://github.com/agda/agda/issues/1435`. On the Agda bug tracker.

L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In *25th International Conference on Automated Deduction (CADE-25)*, 2015.

G. 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.

P. 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.

H. Goguen, C. McBride, and J. McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*. 2006.

J. 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.

C.-K. Hur. Agda with the excluded middle is inconsistent?, 2010. URL `https://lists.chalmers.se/pipermail/agda/2010/001522.html`. On the Agda mailing list.

J.-P. Jouannaud and C. Kirchner. *Solving equations in abstract algebras: A rule-based survey of unification*. Université de Paris-Sud, Centre d'Orsay, Laboratoire de Recherche en Informatique, 1990.

D. R. Licata and M. Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *28th Annual IEEE/ACM Symposium on Logic in Computer Science*, 2013.

Z. Luo. *Computation and reasoning: a type theory for computer science*, volume 11 of *International Series of Monographs on Computer Science*. 1994.

P. Martin-Löf. *Intuitionistic type theory*. Number 1 in Studies in Proof Theory. 1984.

C. McBride. Towards dependent pattern matching in LEGO. TYPES meeting, 1998a.

C. McBride. Inverting inductively defined relations in LEGO. In *Types for Proofs and Programs*, 1998b.

C. McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, 2000.

C. McBride. Elimination with a motive. In *Types for proofs and programs*, 2002.

C. McBride. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, 2005.

C. McBride, H. Goguen, and J. McKinna. A few constructions on constructors. In *Types for Proofs and Programs*, 2006.

J. McKinna and F. N. Forsberg. The encode-decode method, relationally. In *Types for proofs and programs*, 2015.

A. 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.

J.-F. Monin. Proof trick: Small inversions. In *Second Coq Workshop*, 2010.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

D. 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.

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

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

A. Vezzosi. Heterogeneous equality incompatible with univalence even – without-K, 2015. URL `https://github.com/agda/agda/issues/1408`. On the Agda bug tracker.

B. Ziliani and M. Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *ACM SIGPLAN International Conference on Functional Programming*, 2015.