JFP **28**, e12, 55 pages, 2018.
doi:10.1017/S095679681800014X

Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory

JESPER COCKX

Department of Computer Science and Engineering, Chalmers and Gothenburg University, Gothenburg, Sweden
(*e-mail:* jesper@sikanda.be)

DOMINIQUE DEVRIESE

Computer Science, KULeuven, Heverlee, Belgium
(*e-mail:* dominique.devriese@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, standard unification algorithms implicitly rely on principles such as *uniqueness of identity proofs* and *injectivity of type constructors*. These principles are inadmissible in many type theories, particularly in the new and promising branch known as homotopy type theory. As a result, programs and proofs in these new theories cannot make use of dependent pattern matching or other techniques relying on unification, and are as a result much harder to write, modify, and understand. This paper proposes a proof-relevant 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 soundness 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, such as rules for eta-equality of record types and higher dimensional unification rules for solving equations between equality proofs. Using our framework, we implemented a complete overhaul of 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 substantial number of bugs in the process. In the future, we may also want to integrate new principles with pattern matching, for example, the higher inductive types introduced by homotopy type theory. Our framework also provides a solid basis for such extensions to be built on.

1 Introduction

Unification is a generic method for solving symbolic equations algorithmically. It is a fundamental algorithm used in many areas in computer science, such as logic programming, type inference, term rewriting, automated theorem proving, and natural language processing. In particular, type checkers for languages with dependent pattern matching (Coquand, 1992) such as Agda (Norell, 2007), the

equations package for Coq (Sozeau, 2010), Idris (Brady, 2013), and Lean (de Moura *et al.*, 2015) use first-order unification to determine whether a set of constructors covers all possible cases.[1]

Although first-order unification is well understood in the untyped or simply typed setting, its interaction with dependent types has been mysterious so far. In particular, some standard unification rules are no longer valid in the presence of universes and indexed datatypes, two common features of dependently typed languages. Examples of how this can go wrong follow later in this introduction, but we start with an example where first-order unification works as intended.

Example 1. *Consider the type of length-indexed vectors* $\texttt{Vec}\ A\ n$ *(i.e. the type of vectors containing* n *elements of type* A*), where* $\texttt{nil} : \texttt{Vec}\ A\ \texttt{zero}$ *is the empty vector and* $\texttt{cons}\ n\ x\ xs$ *is the vector with head* $x : A$ *and tail* $xs : \texttt{Vec}\ A\ n$*. We can define a safe tail function on vectors by dependent pattern matching as follows:*

$$\begin{aligned} &\texttt{tail} : (n : \mathbb{N}) \to \texttt{Vec}\ A\ (\texttt{suc}\ n) \to \texttt{Vec}\ A\ n \\ &\texttt{tail}\ .m\ (\texttt{cons}\ m\ x\ xs) = xs \end{aligned} \tag{1}$$

The function $\texttt{tail}$ *needs only be defined in the case for* $(\texttt{cons}\ n\ x\ xs)$*: the case for* $\texttt{nil}$ *is impossible because unification of* $\texttt{zero}$ *(the length of* $\texttt{nil}$*) with* $\texttt{suc}\ n$ *reports an absurdity. This is all the better because there is no way to take the tail of an empty vector! In the cases where unification succeeds, it can also teach us something extra about the type of the right-hand side. For example, in the remaining case* $\texttt{tail}\ .m\ (\texttt{cons}\ m\ x\ xs)$*, unification of* $\texttt{suc}\ m$ *with* $\texttt{suc}\ n$ *tells us that* n *must be equal to* m*, as indicated by the so-called* inaccessible pattern *.m. This method of solving equations to either gain more information about the type of the right-hand side or to derive an absurdity is called* specialization by unification *(Goguen, McBride, and McKinna, 2006).*

In a language that has 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, we 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 is allowed to delete reflexive equations of the form $u = u$, 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, the unification algorithm can 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 pair constructor $\_,\_$, an equation $(A, a) = (B, b)$ can be simplified to $A = B$ and $a = b$, but the type of the second equation is now heterogeneous since $a : A$ and $b : B$. Because traditional unification algorithms only look at the syntax of the

[1] They also use a different *higher order* unification algorithm to solve constraints and derive the values of implicit arguments, but this is not the focus of this paper.

terms they are trying to unify, they cause problems when applied to heterogeneous equalities.

Example 2. *Consider the equation* $(\texttt{Bool}, \texttt{true}) = (\texttt{Bool}, \texttt{false})$ *of type* $\Sigma_{A:\texttt{Set}} A$. *By injectivity of the constructor* $\_,\_$, *we can simplify this equation to the two equations* $\texttt{Bool} = \texttt{Bool}$ *and* $\texttt{true} = \texttt{false}$ *and then 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 (Streicher, 1993). In a univalent theory such as HoTT (The Univalent Foundations Program, 2013), it is actually possible to prove that* $(\texttt{Bool}, \texttt{true}) = (\texttt{Bool}, \texttt{false})$ *of type* $\Sigma_{A:\texttt{Set}} A$ *under the equivalence between* Bool *and itself swapping* true *and* false, *refuting the use of injectivity above. So the naive injectivity rule above cannot be used in such a theory.*

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* $\texttt{Set} \times \texttt{Bool}$ *(defined as* $\Sigma_{\_:\texttt{Set}} \texttt{Bool}$*). In this case, it* is *possible to derive an absurdity, even in a univalent theory. However, a unification algorithm can never distinguish between these two equations unless it takes their types into account.*

Example 3. *Suppose we define two copies* $\texttt{Bool}_1$ *and* $\texttt{Bool}_2$ *of the boolean type with constructors* $\texttt{true}_1, \texttt{false}_1$ *and* $\texttt{true}_2, \texttt{false}_2$, *respectively, then it is unsound to apply the conflict rule on the (heterogeneous) equation* $\texttt{true}_1 = \texttt{false}_2$. *Doing so would allow us to prove that* $\texttt{Bool}_1 \not\equiv_{\texttt{Set}} \texttt{Bool}_2$, *again contradicting univalence.*

Example 4. *The problem is not limited to theories that do not support UIP, either. Problems can also occur when we use a naive injectivity rule for constructors of indexed datatypes. Let* A *be an arbitrary type and* $\texttt{Singleton} : A \to \texttt{Set}$ *be an indexed datatype with one constructor* $\texttt{sing} : (x : A) \to \texttt{Singleton}\ x$ *and consider the unification problem* $(\texttt{Singleton}\ s, \texttt{sing}\ s) = (\texttt{Singleton}\ t, \texttt{sing}\ t)$. *If we allow the injectivity rule to simplify* $\texttt{sing}\ x = \texttt{sing}\ y$ *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. *In general, injectivity of type constructors is an undesirable property because it is not only incompatible with the law of the excluded middle (Theorem 93), but also with univalence (Theorem 92) and with an impredicative universe of propositions (Miquel, 2010). In particular, if we let* $A = \texttt{Set} \to \texttt{Set}$ *in the above example, then the injectivity of the* Singleton *type constructor allows us to refute the law of the excluded middle.*

The unification algorithm used by Agda 2.4 (and older) for checking definitions by dependent pattern matching contains a number of restrictions to avoid bad unification steps like in the above examples. One of these restrictions is to not delete equations of the form $u = u$ if the theory does not support UIP (Cockx *et al.*, 2016a). The rule for simplifying equations of the form $\texttt{c}\ u_1\ \ldots\ u_n = \texttt{c}\ v_1\ \ldots\ v_n$ is also restricted in case c is a constructor of an indexed datatype. However, these *ad-hoc* restrictions make the unification algorithm hard to prove correct, modify, or extend.

Contributions. In this paper, we give a *typed* and *proof-relevant* account of the first-order unification algorithm used for checking definitions by dependent pattern

matching, in order to solve the problems with untyped unification in general and put unification in type theory back on a solid theoretical foundation. Concretely, we make the following contributions:

- We give a new representation of unification rules and most general unifiers in a dependently typed setting as *equivalences* between solution spaces represented by telescopic systems of equations, serving as evidence of their soundness (Section 3). This allows us to use type information to decide when a unification rule is applicable and avoid the use of any axioms such as UIP.
- We show how the standard first-order unification rules used by Goguen et al. (2006) can be implemented as equivalences (Section 4). We extend these rules to the case of indexed families of datatypes. These rules work on heterogeneous equations and can solve multiple equations at once, making them more general than the ones in our previous work (Cockx *et al.*, 2014). We also show how to extend the unification algorithm with rules for η-equality of record types.
- To guarantee good computational properties of the unifiers produced by our unification algorithm, we introduce the notion of a *strong unification rule* (Section 5). We show that all the unification rules used by our algorithm are strong ones, except for the (optional) deletion rule.
- We show how to make the injectivity rule for indexed datatypes more general by a technique called *higher dimensional unification* (Section 6). In particular, this technique formalizes the concept of forced constructor arguments, a heuristic that allows unification to skip certain constructor arguments if they are determined by the type of the constructor (Corollary 67).
- We reimplement the unification algorithm used by Agda for pattern matching on indexed families of datatypes based on our framework for proof-relevant unification, eliminating the previous *ad-hoc* restrictions and fixing a number of bugs in the process (Section 8). This new unification algorithm has been released as part of Agda version 2.5.1.[2]

This paper is based on the work of two conference papers (Cockx *et al.*, 2016a; Cockx and Devriese, 2017), as well as the first author's PhD thesis (Cockx, 2017). Compared to the conference papers, we made the following additions:

- We give a new definition of a strong unifier (Definition 53). Compared the previous definition (Cockx *et al.*, 2016a), this definition is more natural to work with and allows us to prove that functions constructed by unification satisfy the expected computational behaviour (Lemma 56), while it is still satisfied by all the unification rules.
- We prove that lifting a strong unifier results again in a strong unifier (Lemma 73).
- We give a formal proof that (a suitably internalized notion of) a most general unifier is really equivalent to an equivalence (Lemmas 17 and 18).
- The presentation of many examples, definitions, lemmas, and theorems was improved compared to the conference version.

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

$$
\begin{array}{rcll}
\Gamma & ::= & () & \text{(empty context)} \\
& | & \Gamma(x : A) & \text{(context extension)} \\
\\
A, B, u, v & ::= & x & \text{(variable)} \\
& | & u\ v & \text{(application)} \\
& | & \lambda x.\, u & \text{(lambda abstraction)} \\
& | & (x : A) \to B & \text{(dependent function type)} \\
& | & \mathtt{Set}_\ell & \text{(universe } \ell\text{)} \\
& | & \mathtt{D} & \text{(inductive datatype)} \\
& | & \mathtt{c} & \text{(data constructor)} \\
& | & \mathtt{f} & \text{(defined function)}
\end{array}
$$

Fig. 1. The syntax for contexts, types, and terms.

Overview. First, we give an overview of the type theory we work in, including syntax and typing rules for telescopes and telescopic equality (Section 2). We start the paper proper with a general description of our framework for reasoning about unification in a dependently typed setting (Section 3). We phrase the basic unification rules in this framework and show how our algorithm can easily be extended by adding more unification rules (Section 4). We pay special attention to the computational behaviour of unification rules when viewed as terms in type theory (Section 5). To augment the power of the unification rules for indexed datatypes, we introduce a new technique called higher dimensional unification (Section 6). We show that our unification rules are conservative over standard type theory by translating them to the standard datatype eliminators (Section 7). We also discuss the implementation of our unification algorithm in Agda (Section 8). We finish the paper with a discussion of related work (Section 9) and future work (Section 10). The appendix contains a proof of the incompatibility between injective type constructors on one hand and univalence and the law of the excluded middle on the other hand (Appendix A).

2 Preliminaries

We base ourselves on Martin-Löf's Intuitionistic Theory of Types with dependent function types, inductive families, and universes (Martin-Löf, 1972, 1984). However, the results in this paper should be equally applicable in other type theories with inductive families such as the Unified Theory of Dependent Types by Luo (1994) or the calculus of inductive constructions used by Coq. The main reason we do not use these more expressive calculi is because we do not need their additional features, and not using them makes our results more generally applicable.

2.1 Basic syntax and typing rules

We use a syntax closely resembling that of Agda (Figure 1). Types and terms share the same syntactic class. As a convention, types are indicated by capital letters A, B, ... and other terms by small letters u, v,... Aside from the standard type-theoretic constructs, the syntax includes datatypes `D`, constructors `c`, and defined functions `f`.

The set of variables that occur freely in u is indicated by $FV(u)$. Simultaneous substitution $u[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ is defined by simultaneously replacing all free

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

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

Fig. 2. The typing rules for valid contexts.

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

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

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

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

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

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

Fig. 3. The core typing rules including dependent function types $(x : A) \to B$ and an infinite hierarchy of universes $\mathtt{Set}_0, \mathtt{Set}_1, \mathtt{Set}_2, \ldots$.

occurrences of $x_1, \ldots, x_n$ in u by $v_1, \ldots, v_n$, avoiding variable capture by renaming bound variables when necessary.

The basic rules for context validity, typing, and definitional equality are given in Figures 2–4, respectively. In addition to these rules, we assume one rule for each datatype `D`, constructor `c`, and defined function `f`, asserting that the symbol has its declared type in any valid context.

2.2 *Inductive families of datatypes*

Inductive families of datatypes are (dependent) types inductively defined by a number of constructors (Dybjer, 1991). Inductive families can also have *parameters* and *indices*. In the syntax (Figure 1), datatypes are written `D` and constructors `c`.

Example 5. $\mathbb{N}$ *is defined as an inductive datatype with the constructors* `zero` $: \mathbb{N}$ *and* `suc` $: \mathbb{N} \to \mathbb{N}$.

Example 6. `Vec` $A\ n$ *is an inductive family with one parameter* A $:$ `Set`, *one index* $n : \mathbb{N}$, *and two constructors* `nil` $:$ `Vec` A `zero` *and* `cons` $: (n : \mathbb{N}) \to A \to$ `Vec` $A\ n \to$ `Vec` A (`suc` n).

In the original definition of indexed families by Dybjer (1991), parameters are required to occur uniformly everywhere in the definition of the datatype, while indices can vary from constructor to constructor. Agda is less restrictive and also allows parameters to occur non-uniformly in the types of recursive constructor

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

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

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

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

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

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

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

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

Fig. 4. The rules for definitional equality, including rules for β-equality, reflexivity, symmetry, transitivity, and congruence.

arguments, but not in their return types. The work in this paper is valid for both variants of the definition.

The values of the parameters are not arguments to the constructor `c`, not even implicitly. This is intentional: requiring constructors to remember their parameters is impractical from an implementation perspective, so we make sure they are never needed for the unification algorithm described in this paper.

2.3 The identity type

The propositional equality type $x \equiv_A y$ expresses the property that x and y are equal elements of type A (Martin-Löf, 1984). The basic way to prove a propositional equality is by using `refl` : $x \equiv_A x$ (short for reflexivity); for example, `refl` is a proof of `zero` $\equiv_{\mathbb{N}}$ `zero`. The identity type comes equipped with a number of useful reasoning principles besides `refl`:

- `sym` : $\{x\ y : A\} \to x \equiv_A y \to y \equiv_A x$ expresses the symmetry of propositional equality.[3]
- `trans` : $\{x\ y\ z : A\} \to x \equiv_A y \to y \equiv_A z \to x \equiv_A z$ expresses the transitivity of propositional equality.

[3] As in Agda, we use two function spaces $(x : A) \to B$ and $\{x : A\} \to B$ for explicit and implicit functions, respectively. The values of implicit arguments can always be deduced from the types of the other arguments, so we omit them when applying the function. For example, we write `sym` e instead of `sym` $x\ y\ e$. In case we do want to make these arguments explicit, we write them between curly brackets as well, for example, `sym` $\{x\}\ \{y\}\ e$.

- `cong` $: (f : A \to B) \to \{x\ y : A\} \to x \equiv_A y \to f\ x \equiv_A f\ y$ expresses congruence: Applying a function to equal arguments gives equal results.
- `subst` $: (P : A \to \mathtt{Set}) \to \{x\ y : A\} \to x \equiv_A y \to P\ x \to P\ y$ expresses substitution: If two types are equal up to some propositionally equal terms, then we can transport elements from one type to the other.

In particular, if we take $A = \mathtt{Set}$ and $P = \lambda X.\ X$ in the type of `subst`, the result is a function of type $(X\ Y : \mathtt{Set}) \to (X \equiv_{\mathtt{Set}} Y) \to X \to Y$. This function is often called `coerce` because it allows us to coerce a term from one type to another if the types are propositionally equal.

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

$$\mathtt{J} : \{x : A\}(P : (y : A) \to x \equiv_A y \to \mathtt{Set})(p : P\ x\ \mathtt{refl})(y : A)(e : x \equiv_A y) \to P\ y\ e \tag{2}$$

This rule is a generalization of `subst` where the type P is allowed to depend on the given equality proof. Using only J, it is possible to define `sym`, `trans`, `cong`, and `subst`, and `coerce`.

We will also rely on the concept of pointwise equality between two functions.

Definition 7 (Pointwise equality). *Let $f, g : (x : A) \to B\ x$ be two functions. The pointwise equality type $f \doteq g$ is defined as $(x : A) \to f\ x \equiv_{B\ x} g\ x$. Similarly, if $\sigma, \tau : \Delta \to \Gamma$ are two telescope mappings, then $\sigma \doteq \tau$ is defined as $(\bar{x} : \Delta) \to \sigma\ \bar{x} \equiv_\Gamma \tau\ \bar{x}$.*

In case we have access to functional extensionality, pointwise equality becomes equivalent with regular propositional equality. However, we do not rely on functional extensionality for any of the work in this paper.

2.4 Equivalences

The central concept we use to represent unification rules in this paper is the notion of an *equivalence*. We use Definition 4.3.1 from The Univalent Foundations Program (2013):

Definition 8. *A function $f : A \to B$ is an* equivalence *if we have two functions $g_1 : B \to A$ and $g_2 : B \to A$ and proofs that they are, respectively, a left and a right inverse of f (i.e. terms of type $(x : A) \to g_1\ (f\ x) \equiv_A x$ and $(y : B) \to f\ (g_2\ y) \equiv_B y$). Two types A and B are* equivalent *if there exists an equivalence from A to B. The type of all equivalences $f : A \to B$ is written as $A \simeq B$.*

If $f : A \simeq B$ is an equivalence, then we also write f for the function $f : A \to B$. We write `linv` f for the function g_1 (for the **l**eft **inv**erse of f), and `isLinv` f for the proof of $(x : A) \to$ `linv` $f\ (f\ x) \equiv_A x$. Similarly, `rinv` f stands for the function g_2 and `isRinv` f for the proof of $(y : B) \to f\ (\mathtt{rinv}\ f\ y) \equiv_B y$. For many equivalences, `linv` f and `rinv` f are definitionally equal. In that case, we write f^{-1} for their common value.

The notion of equivalence plays a central role in Voevodsky's univalence axiom. However, our work does not require any primitives on the top of the basic intuitionistic type theory (such as univalence). In fact, our work can be equally

$$\frac{\Gamma\ \mathbf{context}}{\Gamma \vdash ()\ \mathbf{telescope}}\ (\text{Tel-empty})$$

$$\frac{\Gamma \vdash A : \mathtt{Set}_\ell \qquad \Gamma(x : A) \vdash \Delta\ \mathbf{telescope}}{\Gamma \vdash (x : A)\Delta\ \mathbf{telescope}}\ (\text{Tel-extend})$$

$$\frac{\Gamma\ \mathbf{context}}{\Gamma \vdash () : ()}\ (\text{List-empty})$$

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

Fig. 5. The typing rules for telescopes.

well understood without any knowledge of HoTT, and is still useful in a setting that assumes entirely different axioms (such as the law of the excluded middle from classical logic).

2.5 Telescopes

A telescope is a list of typed variable bindings where each type can depend on the previous variables (de Bruijn, 1991). For example, $(m : \mathbb{N})(p : m \equiv_{\mathbb{N}} \mathtt{zero})$ is a telescope of length 2. Telescopes are much like contexts in the sense that they consist of a sequence of variable typings of the form $(x : A)$. However, they are used for different purposes so it is best to keep the two concepts separate. While contexts grow to the right, telescopes grow to the left. One way to think about a telescope is as the *tail* of a context: While a context must always be closed, a telescope can contain free variables from an ambient context, and the telescope can be added to that context to produce a new, extended context.

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

Telescopes are used as the type of a list of terms. A list of terms is indicated by a bar above the letter, for example, $\bar{t} = (\mathtt{zero}; \mathtt{refl}) : (m : \mathbb{N})(p : m \equiv_{\mathbb{N}} \mathtt{zero})$. We also write $()$ for the empty list of terms. The typing rules for telescopes and lists of terms are given in Figure 5.

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

- ... as an extension to the context: $\Gamma\Delta$ is defined by $\Gamma() := \Gamma$ and $\Gamma((x : A)\Delta) := (\Gamma(x : A))\Delta$. In particular, if Δ is a valid telescope in the empty context, then Δ can be used as the context $()\Delta$.
- ... as the names of the variables of a parallel substitution: $u[\Delta \mapsto \bar{v}]$ is defined by substituting the values $\bar{v}$ for the variables of Δ in u.
- ... as the argument types of an iterated function type: $\Delta \to B$ is defined by $() \to B := B$ and $(x : A)\Delta \to B := (x : A) \to (\Delta \to B)$.
- ... in the definition of an iterated lambda abstraction: $\lambda\Delta.\ u$ is defined by $\lambda().\ u := u$ and $\lambda((x : A)\Delta).u := \lambda x.\ (\lambda\Delta.\ u)$.

... as a list of the variables in the telescope: $f\ \Delta$ is defined by $f\ () := f$ and $f\ ((x : A)\Delta) := (f\ x)\ \Delta$.

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

If $\mathtt{D}$ is a datatype with indices Ξ, we write $\overline{\mathtt{D}}$ for the telescope $(\bar{u} : \Xi)(x : \mathtt{D}\ \bar{u})$, for example, $\overline{\mathtt{Vec}\ A} = (n : \mathbb{N})(x : \mathtt{Vec}\ A\ n)$.

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

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

Example 9. *Suppose* $\Delta = (k : \mathbb{N})$ *and* $\Delta' = (m\ n : \mathbb{N})$ *and let* $f\ k = (\mathtt{zero}; \mathtt{suc}\ k)$. *We have* $\Delta' \vdash m{+}n : \mathbb{N}$, *so applying the substitution* $[\Delta' \mapsto f\ \Delta] = [m \mapsto \mathtt{zero}; n \mapsto \mathtt{suc}\ k]$ *gives us* $\Delta \vdash \mathtt{zero} + \mathtt{suc}\ k : \mathbb{N}$.

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

2.6 Heterogeneous and telescopic equality

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

- Turning a proof of heterogeneous equality between elements of the same type into a homogeneous one requires UIP. So in a theory without UIP (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. Example 2 shows that both cases are possible, and that it is essential to know the difference!

- Finally, it is unsound to postpone an equation and continue with the next one when working with heterogeneous equality, since this allows us to prove things such as injectivity of certain type constructors (Example 4).

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

Definition 10. *Let* $e : s \equiv_A t$ *and* $P : (x : A) \to \mathtt{Set}$. *We define the type* $u \equiv^e_P v$ *of equality proofs between* $u : P\ s$ *and* $v : P\ t$ laying over e *by*

$$u \equiv^e_P v = (\mathtt{subst}\ P\ e\ u) \equiv_{(P\ t)} v \tag{3}$$

In practice, the exact definition of $u \equiv^e_P v$ does not have much impact, what is important is that $(u \equiv^{\mathtt{refl}}_P v) = (u \equiv_{P\ s} v)$ whenever $s = t$. An alternative definition would be to define $u \equiv^e_P v$ as a new datatype indexed over s, t, e, u, and v with a single constructor $\mathtt{refl} : u \equiv^{\mathtt{refl}}_P u$. Yet, another alternative definition would be to define the type $u \equiv^e_P v$ by matching on e, giving $(u \equiv^{\mathtt{refl}}_P v) = u \equiv_{P\ s} v$ in case e is $\mathtt{refl}$. We prefer our definition to these more symmetric alternatives because it does not require auxiliary datatypes or large eliminations.

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

Using this notion of an equality proof laying over another, we can define a version of $\mathtt{cong}$ that works for dependent functions:

$$\begin{aligned} &\mathtt{dcong} : (f : (x : A) \to B\ x) \to \{x\ y : A\} \to (e : x \equiv_A y) \to f\ x \equiv_{B\ e} f\ y \\ &\mathtt{dcong}\ f\ \mathtt{refl} = \mathtt{refl} \end{aligned} \tag{4}$$

Telescopic equality is defined as follows:

Definition 11. *Let* Δ *be a telescope and* $\Gamma \vdash \bar{s}, \bar{t} : \Delta$. *We define a new telescope* $(\bar{e} : \bar{s} \equiv_\Delta \bar{t})$ *called the* telescopic equality *between* $\bar{s}$ *and* $\bar{t}$ *inductively on the length of* Δ *by* $() \equiv_{()} () = ()$ *and*

$$(e; \bar{e} : s; \bar{s} \equiv_{(x:A)\Delta} t; \bar{t}) = (e : s \equiv_A t)(\bar{e} : \bar{s} \equiv_{\Delta[x \mapsto e]} \bar{t}) \tag{5}$$

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

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

Lemma 12. *We have the telescopic equality eliminator*

$$\bar{\mathtt{J}} : (P : (\bar{s} : \Delta) \to \bar{r} \equiv \bar{s} \to \mathtt{Set}_i) \to P\ \bar{r}\ \overline{\mathtt{refl}} \to (\bar{s} : \Delta) \to (\bar{e} : \bar{r} \equiv \bar{s}) \to P\ \bar{s}\ \bar{e} \tag{6}$$

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

$$\begin{aligned}
\overline{\mathsf{J}}\ P\ p\ ()\ \ \ ()\ \ \ &= p \\
\overline{\mathsf{J}}\ P\ p\ (s;\bar{s})\ (e;\bar{e}) &= \mathsf{J}\ (\lambda s; e.\ (\bar{s} : \Delta)(\bar{e} : \bar{r} \equiv \bar{s}) \to P\ (s;\bar{s})\ (e;\bar{e})) \\
&\qquad (\lambda \bar{s}; \bar{e}.\ \overline{\mathsf{J}}\ (\lambda \bar{s}; \bar{e}.\ P\ (r;\bar{s})\ (\mathtt{refl};\bar{e}))\ p\ \bar{e}) \\
&\qquad e\ \bar{s}\ \bar{e}
\end{aligned} \tag{7}$$

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

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

$$\begin{aligned}
\overline{\mathtt{subst}} &: (P : \Delta \to \mathtt{Set}_\ell) \to \{\bar{u}\ \bar{v} : \Delta\} \to \bar{u} \equiv_\Delta \bar{v} \to P\ \bar{u} \to P\ \bar{v} \\
\overline{\mathtt{cong}} &: (f : \Delta \to T) \to \{\bar{u}\ \bar{v} : \Delta\} \to \bar{u} \equiv_\Delta \bar{v} \to f\ \bar{u} \equiv_T f\ \bar{v} \\
\overline{\mathtt{dcong}} &: (f : (\bar{x} : \Delta) \to T\ \bar{x}) \to \{\bar{u}\ \bar{v} : \Delta\} \to (\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to f\ \bar{u} \equiv_{T\ \bar{e}} f\ \bar{v}
\end{aligned} \tag{8}$$

3 Unification in dependent type theory

In this section, we describe our new framework for unification of dependently typed data. First, we represent the input of the unification problem by a *telescopic equality* where each type in this telescope corresponds to one equation of the unification problem (Section 3.1). The output of the unification algorithm is then an equivalence between the original telescope of equations and a trivial one (Section 3.2). This equivalence contains not only the substitution computed by the unification algorithm, but also *evidence* that the output is sound. The unification algorithm itself works by successively applying unification rules, which are represented by equivalences between two telescopes of equations (Section 3.3). The aggregate equivalence produced by unification can be used for *specialization by unification*, an essential part of the translation of definitions by dependent pattern matching to eliminators (Section 3.4).

3.1 Unification problems as telescopes

To represent equations internally, we use the propositional equality type $x \equiv_A y$. For example, the equation $\mathtt{suc}\ m = \mathtt{suc}\ n$ is represented by the type $\mathtt{suc}\ m \equiv_{\mathbb{N}} \mathtt{suc}\ n$. In general, a unification problem can consist of multiple equalities and the type of an equality may depend on the solution of the previous equalities. To keep track of these dependencies, we give a type to the list of equations in the form of a telescope. By the nature of a telescope, the type of each equation can depend on the previous equations.

Definition 13. *A* unification problem *is a telescope of the form* $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. *The variables in* Γ *are called the* flexible variables.

Example 14. *A unification problem between two vectors can be represented by the telescope*

$$(m\ n : \mathbb{N})(x\ y : A)(xs : \texttt{Vec}\ A\ m)(ys : \texttt{Vec}\ A\ n)$$
$$(e_1 : \texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n)(e_2 : \texttt{cons}\ m\ x\ xs \equiv_{\texttt{Vec}\ A\ e_1} \texttt{cons}\ n\ y\ ys) \tag{9}$$

3.2 Unifiers as equivalences

Traditionally, a unifier for a unification problem $\bar{u} = \bar{v}$ is defined as a substitution σ such that $\bar{u}\sigma$ and $\bar{v}\sigma$ are equal. So how do we translate this definition to type theory? Consider a unification problem of the form $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. We could represent a unifier as a telescope mapping $\sigma : \Gamma' \to \Gamma$ satisfying $\bar{u}[\Gamma \mapsto \sigma\ \Gamma] = \bar{v}[\Gamma \mapsto \sigma\ \Gamma]$, but then the soundness 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 15. *Let Γ and Δ be telescopes and $\bar{u}$ and $\bar{v}$ be lists of terms such that $\Gamma \vdash \bar{u}, \bar{v} : \Delta$. We define a* unifier *of $\bar{u}$ and $\bar{v}$ as a telescope mapping $\sigma : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ for some Γ'.*

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

Example 16. *The telescope mapping $\sigma : (k : \mathbb{N}) \to (k\ l : \mathbb{N})(e : \texttt{suc}\ k \equiv_{\mathbb{N}} \texttt{suc}\ l)$ defined by $\sigma = \lambda k.k;k;\texttt{refl}$ is a unifier of $\texttt{suc}\ k$ and $\texttt{suc}\ l$. The evidence here is $\texttt{refl} : \texttt{suc}\ k = \texttt{suc}\ k$, proving that $\texttt{suc}\ k$ and $\texttt{suc}\ l$ become equal under the substitution $[k \mapsto k; l \mapsto k]$.*

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

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

Lemma 17. *Let Γ and Γ' be telescopes and $\sigma : \Gamma' \to \Gamma$. The following two statements are equivalent:*

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

Proof

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

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

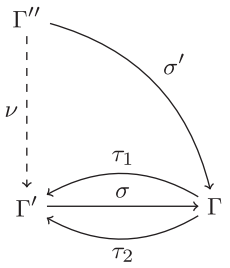

Fig. 6. Lemma 17 allows us to construct a right inverse τ_1 to σ from the existence of the telescope mapping ν, while Lemma 18 gives us a left inverse τ_2 from its uniqueness.

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

Lemma 18. *Let Γ and Γ' be telescopes and let $\sigma : \Gamma' \to \Gamma$. The following two statements are equivalent:*

- *For any telescope Γ'' and $\sigma' : \Gamma'' \to \Gamma$, there exists* at most *one $\nu : \Gamma'' \to \Gamma'$ such that $\sigma' \doteq \sigma \circ \nu$.*
- *There exists a $\tau_2 : \Gamma \to \Gamma'$ such that $\tau_2 \circ \sigma \doteq \texttt{id}$.*

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

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

The proofs of Lemmas 17 and 18 are illustrated in Figure 6. If we replace the telescope Γ by a unification problem $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$, then Lemmas 17 and 18 together give us that $\sigma : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to \Gamma'$ is a most general unifier if and only if it is an *equivalence* between Γ' and $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. This brings us to the following definition of a most general unifier:

Definition 19. *Let Γ and Δ be telescopes and $\bar{u}$ and $\bar{v}$ be lists of terms such that $\Gamma \vdash \bar{u}, \bar{v} : \Delta$. Then a* most general unifier *of $\bar{u}$ and $\bar{v}$ is an equivalence $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ for some telescope Γ'.*

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

The definition of a most general unifier does not prevent us from choosing $\Gamma' = \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ and $f = \mathtt{id}$. In fact, this is a valid (if trivial) most general unifier from a logical point of view.

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

Definition 20. *Let* Γ *and* Δ *be telescopes and* $\bar{u}$ *and* $\bar{v}$ *be lists of terms such that* $\Gamma \vdash \bar{u}, \bar{v} : \Delta$. *A* disunifier *of* $\bar{u}$ *and* $\bar{v}$ *is an equivalence* $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$.

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

3.3 The unification algorithm

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

Definition 21. *A* positive unification rule *is an equivalence of the form* $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$.

For example, the unification rule for injectivity of the $\mathtt{suc}$ constructor for $\mathbb{N}$ is

$$\mathtt{injectivity}_{\mathtt{suc}} : (e : \mathtt{suc}\ m \equiv \mathtt{suc}\ n) \simeq (e : m \equiv n) \tag{10}$$

Another important example of a positive unification rule is the solution rule used to solve equations where one side is a variable:

$$\mathtt{solution} : (x : A)(e : x \equiv_A t) \simeq () \tag{11}$$

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

Definition 22. *A* negative unification rule *is an equivalence of the form* $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$.

For example, the unification rule for conflict between $\mathtt{true}$ and $\mathtt{false}$ is

$$\mathtt{conflict}_{\mathtt{true},\mathtt{false}} : (\mathtt{true} \equiv_{\mathtt{Bool}} \mathtt{false}) \simeq \bot \tag{12}$$

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

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

We do not yet give an explicit strategy on which rule to apply in a specific situation. This leaves more freedom to the implementation to choose which rule to try first. When we discuss our implementation in Section 8, we give one concrete strategy.

Example 23. *Consider the unification problem consisting of flexible variables* $k\ l : \mathbb{N}$ *and a single equation between* $\mathtt{suc}\ k$ *and* $\mathtt{suc}\ l$*. First, we simplify the equation by applying the equivalence* $\mathtt{injectivity}_{\mathtt{suc}} : (e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l) \simeq (e : k \equiv_{\mathbb{N}} l)$*. 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 ends in a positive success.*

We write down the unification process as a series of telescopes (representing unification problems) with equivalences between them (representing the individual unification steps). In each step, we underline the variables and equations that are being solved or simplified.

$$\begin{aligned}
&(k\ l : \mathbb{N})(\underline{e} : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l) \\
&\quad \simeq (k\ \underline{l} : \mathbb{N})(\underline{e} : k \equiv_{\mathbb{N}} l) \\
&\quad \simeq (k : \mathbb{N})
\end{aligned} \tag{13}$$

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 *to be* k *and* e *to be* `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 aggregate unifier* $f^{-1} : (k : \mathbb{N}) \to (k\ l : \mathbb{N})(\underline{e} : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l)$ *here is* $\lambda k.k;k;\mathtt{refl}$ *(since* $\mathtt{cong}\ \mathtt{suc}\ \mathtt{refl} = \mathtt{refl}$*).*

Before we continue with the general form of the unification rules in the next section, we first give three easy but useful manipulations on equivalences (and hence on unification rules) that allow us to postpone and reorder equations. These principles are used when we want to apply a unification rule, but the problem contains some additional variables or equations that are not mentioned in the rule. For example, in the second step of Example 23, the solution rule did not affect the variable k. In what follows, we often make use of these manipulations implicitly.

Lemma 24. *If we have an equivalence* $f : \Delta \simeq \Delta'$ *where* Δ *and* Δ' *possibly contain free variables from a telescope* Γ*, then we also have an equivalence* $f_\Gamma : \Gamma\Delta \simeq \Gamma\Delta'$*.*

Lemma 25. *If we have an equivalence* $f : \Gamma \simeq \Gamma'$ *and a telescope* Δ *possibly containing free variables from* Γ*, then we also have an equivalence* $f^\Delta : \Gamma\Delta \simeq \Gamma'\Delta'$*, where* $\Delta' = \Delta[\Gamma \mapsto \mathtt{linv}\ f\ \Gamma']$*.*

Lemma 26. *If we have a telescope* Γ*, and* Γ' *is a reordering of the variable bindings in* Γ *that preserves the order of dependencies, then we have an equivalence* $f : \Gamma \simeq \Gamma'$*.*

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

3.4 Specialization by unification

When performing case analysis on a variable from an inductive family, the typechecker of a dependently typed language needs to determine which constructors can occur in a given position and how the variables need to be instantiated for the pattern to be well-typed. To do this, it applies unification to the indices of the datatype in question. If unification determines that there can be no such substitution, then we can skip the case for the corresponding constructor. This method of solving equations to either gain more information about the type of the right-hand side or to derive an absurdity is called *specialization by unification.*

Specialization by unification allows us to construct functions of the form

$$m : (\bar{x} : \Gamma)(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to T\ \bar{x}\ \bar{e} \tag{14}$$

It can be seen as a generic method of constructing an *inversion principle* (McBride, 1998b). It is also a core component in the translation of definitions by pattern matching to eliminators (Goguen et al., 2006; Cockx, Devriese, and Piessens, 2016b).

Definition 27 (Specialization by unification). *Consider a problem of the form* $m : (\bar{x} : \Gamma)(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to T\ \bar{x}\ \bar{e}$ *and suppose that unification of* $\bar{u}$ *with* $\bar{v}$ *with* Γ *as flexible variables succeeds either positively or negatively, then we construct the function* m*:*

- *In case the unification succeeds positively with most general unifier* $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$*, we have*

$$\mathtt{isLinv}\ f : (\bar{x} : \Gamma)(\bar{e} : \bar{u} \to f^{-1}\ (f\ \bar{x}\ \bar{e}) \equiv_{\Gamma(\bar{e} : \bar{u}\ \bar{x}; \bar{e})} \tag{15}$$

  Then we define m *by*

$$m\ \bar{x}\ \bar{e} = \overline{\mathtt{subst}}\ T\ (\mathtt{isLinv}\ f\ \bar{x}\ \bar{e})\ (m^s\ (f\ \bar{x}\ \bar{e})) \tag{16}$$

  with the new subgoal of constructing $m^s : (\bar{x}' : \Gamma') \to T\ (\mathtt{linv}\ f\ \bar{x}')$*.*
- *In case the unification succeeds negatively with disunifier* $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \bot$*, then we define* m *by*

$$m\ \bar{x}\ \bar{e} = \mathtt{elim}_\bot\ (T\ \bar{x}\ \bar{e})\ (f\ \bar{x}\ \bar{e}) \tag{17}$$

  with no additional assumptions.

Example 28. *We apply specialization by unification to construct two functions*

$$\begin{aligned} m_{\mathtt{lz}} : (m : \mathbb{N})(k : \mathbb{N})(y : k \leqslant \mathtt{zero}) \to \\ (\mathtt{zero}; m; \mathtt{lz}\ m) \equiv_{(m\ n:\mathbb{N})(x:m\leqslant n)} (k; \mathtt{zero}; y) \to \mathtt{zero} \equiv_{\mathbb{N}} k \end{aligned} \tag{18}$$

and

$$\begin{aligned} m_{\mathtt{ls}} : (m\ n : \mathbb{N})(x : m \leqslant n)(k : \mathbb{N})(y : k \leqslant \mathtt{zero}) \to \\ (\mathtt{suc}\ m; \mathtt{suc}\ n; \mathtt{ls}\ m\ n\ x) \equiv_{(m\ n:\mathbb{N})(x:m\leqslant n)} (k; \mathtt{zero}; y) \to \mathtt{zero} \equiv_{\mathbb{N}} k \end{aligned} \tag{19}$$

In case of $m_{\texttt{lz}}$, *unification of* $\bar{u} = \texttt{zero};m;\texttt{lz}\ m$ *with* $\bar{v} = k;\texttt{zero};y$ *in the context* $\Gamma = (m : \mathbb{N})(k : \mathbb{N})(y : k \leqslant \texttt{zero})$ *results in a positive success with most general unifier* $f : \Gamma(\bar{u} \equiv_{(m\ n:\mathbb{N})(x:m\leqslant n)} \bar{v}) \simeq ()$ *with* $f^{-1}\ () = \texttt{zero};\texttt{zero};\texttt{lz}\ \texttt{zero};\overline{\texttt{refl}}$, *where* $\texttt{refl} : \texttt{zero};\texttt{zero};\texttt{lz}\ \texttt{zero} \equiv_{(m\ n:\mathbb{N})(x:m\leqslant n)} \texttt{zero};\texttt{zero};\texttt{lz}\ \texttt{zero}$. *Specialization by unification gives us the function* $m_{\texttt{lz}}$ *on the condition we can construct* ${m_{\texttt{lz}}}^{s} : \texttt{zero} \equiv_{\mathbb{N}} \texttt{zero}$, *which we can do easily as* ${m_{\texttt{lz}}}^{s} = \texttt{refl}$.

For $m_{\texttt{ls}}$, *unification of* $\bar{u} = \texttt{suc}\ m;\texttt{suc}\ n;\texttt{ls}\ m\ n\ x$ *with* $\bar{v} = k;\texttt{zero};y$ *results in a negative success, so specialization by unification gives us the function* $m_{\texttt{ls}}$ *without any additional assumptions.*

4 Unification rules

In this section, we state the basic unification rules from McBride (1998b) in our framework. We first handle the unification rules for simple datatypes (Section 4.1) before moving on to the more challenging rules for indexed datatypes (Section 4.2). We also show how to extend the unification algorithm with rules for η-equality for record types (Section 4.3).

4.1 The basic unification rules

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

Lemma 29. *For any type* A *and term* $t : A$, *we have an equivalence*

$$\texttt{solution} : (x : A)(e : x \equiv_A t) \simeq () \tag{20}$$

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

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

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

Lemma 30. *For any type* A *that satisfies UIP and any term* $t : A$, *we have an equivalence*

$$\texttt{deletion} : (e : t \equiv_A t) \simeq () \tag{21}$$

Construction of `deletion`. This follows directly from the UIP at type A. □

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

$$\frac{}{t_i\ s_1\ \ldots\ s_k \prec \mathtt{c}\ t_1\ \ldots\ t_n} \qquad \frac{s \prec t_i}{s \prec \mathtt{c}\ t_1\ \ldots\ t_n}$$

Fig. 7. The structural order $\prec$ is used to check termination and to detect cycles during unification.

Lemma 31. *Let* $\mathtt{c} : \Delta_{\mathtt{c}} \to \mathtt{D}$ *be a constructor of the datatype* $\mathtt{D} : \mathtt{Set}_\ell$ *and let* $\bar{x}, \bar{x}' : \Delta_{\mathtt{c}}$. *We have an equivalence*

$$\mathtt{injectivity}_{\mathtt{c}} : (\mathtt{c}\ \bar{x} \equiv_{\mathtt{D}} \mathtt{c}\ \bar{x}') \simeq (\bar{x} \equiv_{\Delta_{\mathtt{c}}} \bar{x}') \tag{22}$$

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

Lemma 32. *Let* $\mathtt{c}_1 : \Delta_1 \to \mathtt{D}$ *and* $\mathtt{c}_2 : \Delta_2 \to \mathtt{D}$ *be two distinct constructors of the datatype* $\mathtt{D} : \mathtt{Set}_\ell$ *and* $\bar{x}_1 : \Delta_1$ *and* $\bar{x}_2 : \Delta_2$. *We have an equivalence*

$$\mathtt{conflict}_{\mathtt{c}_1,\mathtt{c}_2} : (\mathtt{c}_1\ \bar{x}_1 \equiv_{\mathtt{D}} \mathtt{c}_2\ \bar{x}_2) \simeq \bot \tag{23}$$

To state the **cycle** rule, we need the structural order $\prec$ defined in Figure 7. This definition is somewhat different from the one given by Goguen et al. (2006). It describes the same relation in case the left- and right-hand sides are elements of the datatype $\mathtt{D}$, but it enforces that the left- and right-hand sides are actually elements of the datatype. This prevents odd structural orders that are allowed by the original definition. For example, if we have a datatype $\mathtt{D}$ with a constructor $\mathtt{c} : (A \to \mathtt{D}) \to \mathtt{D}$, then the definition by Goguen et al. allows us to derive for any $a : A$ that $a \prec \mathtt{c}\ a \prec \mathtt{c}$, even though a does not occur in $\mathtt{c}$.

Lemma 33. *Let* $\mathtt{D} : \mathtt{Set}_\ell$ *be a datatype and let* $x, t : \mathtt{D}$ *be such that* $x \prec t$. *We have an equivalence*

$$\mathtt{cycle}_{x,t} : (x \equiv_{\mathtt{D}} t) \simeq \bot \tag{24}$$

Once again, the type of the equation should be exactly $\mathtt{D}$. We postpone the proof of this lemma until Section 7.

Example 34. *Consider the sum type* $A \uplus B$ *(where* $A, B : \mathtt{Set}$ *are arbitrary types) with two constructors* $\mathtt{left} : A \to A \uplus B$ *and* $\mathtt{right} : B \to A \uplus B$. *An expression of the form* $\mathtt{left}\ x$ *is never equal to* $\mathtt{right}\ y$, *so any equality between those two terms is equivalent to* $\bot$:

$$(x : A)(y : B)(\underline{e} : \mathtt{left}\ x \equiv_{A \uplus B} \mathtt{right}\ y) \simeq \bot \tag{25}$$

This is exactly the conflict rule between left *and* right.

The type of the equation on the left in Lemmas 31 and 32 should be exactly $\mathtt{D}$, in particular, the constructor $\mathtt{c}$ must be fully applied.

Counterexample 35. *An equation between the constructors* left *and* right *is not always absurd when they are not fully applied. Let* $A = B = \bot$, *then* $(e : \mathtt{left} \equiv_{\bot \to \bot \uplus \bot} \mathtt{right})$ *is not equivalent to* $\bot$. *This is because when viewed as functions of type* $\bot \to \bot \uplus \bot$, *the constructors* $\mathtt{inj}_1$ *and* $\mathtt{inj}_2$ *coincide on all possible inputs (i.e. none). The principle of functional extensionality then tells us that these two functions are equal. So if we would consider this equation to be absurd, we would prohibit ourselves*

$$\begin{aligned}
\texttt{solution} &: (x : A)(e : x \equiv_A u) \simeq () && (\text{where } x \notin FV(u)) \\
\texttt{deletion} &: (e : u \equiv_A u) \simeq () \\
\texttt{injectivity}_{\texttt{c}} &: (\bar{u}[\Delta \mapsto \bar{x}]; \texttt{c}\ \bar{x} \equiv_{\overline{\texttt{D}}} \bar{u}[\Delta \mapsto \bar{x}']; \texttt{c}\ \bar{x}') \simeq (\bar{x} \equiv_\Delta \bar{x}') \\
\texttt{conflict}_{\texttt{c}_1,\texttt{c}_2} &: (\bar{u}_1[\Delta_1 \mapsto \bar{x}_1]; \texttt{c}_1\ \bar{x}_1 \equiv_{\overline{\texttt{D}}} \bar{u}_2[\Delta_2 \mapsto \bar{x}_2]; \texttt{c}_2\ \bar{x}_2) \simeq \bot && (\text{where } \texttt{c}_1 \neq \texttt{c}_2) \\
\texttt{cycle}_{x,t} &: (\bar{u}; x \equiv_{\overline{\texttt{D}}} \bar{v}; t) \simeq \bot && (\text{where } x \prec t)
\end{aligned}$$

Fig. 8. The basic unification rules can be formulated as equivalences.

from having a general rule for functional extensionality in our language, nevertheless a desirable property to have! Wrongly applying the conflict rule in this way led to the problem described by issue #1497 on the Agda bug tracker (Dijkstra, 2015).

4.2 Rules for indexed datatypes

The injectivity, conflict, and cycle rules defined in the previous section all work on regular datatypes, but unification only becomes really interesting once we consider indexed families of datatypes. Where the unification rules that we have seen so far only have a single equation on the left side, the rules for indexed datatypes have a telescope of equations: one equation for each index, and one final equation for the datatype itself.

Lemma 36. *Let* $\texttt{c} : \Delta \to \texttt{D}\ \bar{u}$ *be a constructor of the datatype* $\texttt{D} : \Xi \to \texttt{Set}_\ell$. *Then we have an equivalence*

$$\texttt{injectivity}_{\texttt{c}} : (\bar{u}[\Delta \mapsto \bar{x}]; \texttt{c}\ \bar{x} \equiv_{\overline{\texttt{D}}} \bar{u}[\Delta \mapsto \bar{x}']; \texttt{c}\ \bar{x}') \simeq (\bar{x} \equiv_\Delta \bar{x}') \tag{26}$$

where $\bar{x}, \bar{x}' : \Delta$ *and* $\texttt{injectivity}_{\texttt{c}}{}^{-1}\ \bar{e} = \overline{\texttt{dcong}}\ (\lambda \bar{x}.\ \bar{u}; \texttt{c}\ \bar{x})\ \bar{e}$.

Lemma 37. *Let* $\texttt{c}_1 : \Delta_1 \to \texttt{D}\ \bar{u}_1$ *and* $\texttt{c}_2 : \Delta_2 \to \texttt{D}\ \bar{u}_2$ *be two distinct constructors of the datatype* $\texttt{D} : \Xi \to \texttt{Set}_\ell$. *Then we have an equivalence*

$$\texttt{conflict}_{\texttt{c}_1,\texttt{c}_2} : (\bar{u}_1[\Delta_1 \mapsto \bar{x}_1]; \texttt{c}_1\ \bar{x}_1 \equiv_{\overline{\texttt{D}}} \bar{u}_2[\Delta_2 \mapsto \bar{x}_2]; \texttt{c}_2\ \bar{x}_2) \simeq \bot \tag{27}$$

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

Lemma 38. *Let* $\texttt{D} : \Xi \to \texttt{Set}_\ell$ *be a datatype and let* $(\bar{u}; x), (\bar{v}; t) : \overline{\texttt{D}}$ *be such that* $x \prec t$. *Then we have an equivalence*

$$\texttt{cycle}_{x,t} : (\bar{u}; x \equiv_{\overline{\texttt{D}}} \bar{v}; t) \simeq \bot \tag{28}$$

Again, we postpone the proof of this lemma until Section 7. The unification rules are summarized in Figure 8. To these basic rules, we will add rules for η-equality for record types (Section 4.3), generalized rules for conflict and acyclicity (Section 6.1), and higher dimensional unification (Section 6.2). We give a complete list of the unification rules with all these additions in Figure 9.

Example 39. *Consider the indexed datatype* $\texttt{Vec}\ A : \mathbb{N} \to \texttt{Set}$ *with the two constructors* $\texttt{nil} : \texttt{Vec}\ A\ \texttt{zero}$ *and* $\texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)$.

$$
\begin{aligned}
\texttt{solution} &: (x : A)(e : x \equiv_A u) \simeq () && \text{(where } x \notin FV(u)\text{)}\\
\texttt{deletion} &: (e : u \equiv_A u) \simeq ()\\
\texttt{injectivity}''^{f}_{\texttt{c}} &: (\bar{e} : \bar{s}_1; \texttt{c}\ \bar{t}_1 \equiv_{\Phi(z:\texttt{D}\ \bar{v})} \bar{s}_2; \texttt{c}\ \bar{t}_2) \simeq (\bar{e}' : f\ \bar{s}_1\ \bar{t}_1\ \overline{\texttt{refl}} \equiv_{\Delta'} f\ \bar{s}_2\ \bar{t}_2\ \overline{\texttt{refl}})\\
&\qquad \text{(where } \texttt{c} : \Delta \to \texttt{D}\ \bar{u} \text{ and } f : \Phi\Delta(\bar{p} : \bar{u} \equiv_{\Xi} \bar{v}) \simeq \Delta'\text{)}\\
\texttt{conflict}'_{\texttt{c}_1,\texttt{c}_2} &: \left(\bar{s}_1; \texttt{c}_1\ \bar{t}_1 \equiv_{\Phi(x:\texttt{D}\ \bar{v})} \bar{s}_2; \texttt{c}_2\ \bar{t}_2\right) \simeq \bot && \text{(where } \texttt{c}_1 \neq \texttt{c}_2\text{)}\\
\texttt{cycle}'_{t_1,t_2} &: \left(\bar{s}_1; t_1 \equiv_{\Phi(x:\texttt{D}\ \bar{v})} \bar{s}_2; t_2\right) \simeq \bot && \text{(where } t_1 \prec t_2\text{)}\\
\eta\texttt{var}_{\texttt{R}} &: (r : \texttt{R}) \simeq (f_1 : A_1) \ldots (f_n : A_n\ f_1\ \ldots\ f_{n-1})\\
\eta\texttt{eq}_{\texttt{R}} &: (e : r \equiv_R s) \simeq (e_1 : \texttt{f}_1\ r \equiv_{A_1} \texttt{f}_1\ s) \ldots (e_n : \texttt{f}_\texttt{n}\ r \equiv_{A_n\ e_1\ \ldots\ e_{n-1}} \texttt{f}_\texttt{n}\ s)
\end{aligned}
$$

Fig. 9. The complete list of rules of the unification algorithm.

The injectivity rule for cons *gives us the following equivalence:*

$$
\begin{aligned}
&(\texttt{suc}\ m; \texttt{cons}\ m\ x\ xs \equiv_{\overline{\texttt{Vec}\ A}} \texttt{suc}\ n; \texttt{cons}\ n\ y\ ys)\\
&\quad \simeq (m; x; xs \equiv_{(n:\mathbb{N})(x:A)(xs:\texttt{Vec}\ A\ n)} n; y; ys)
\end{aligned}
\tag{29}
$$

This rule not only simplifies the equation between the two cons *constructors, but simultaneously simplifies the equation between the indices* suc *m and* suc *n. Now let us see how this rule works in action:*

$$
\begin{aligned}
&(m\ n : \mathbb{N})(x\ y : A)(xs : \texttt{Vec}\ A\ m)(ys : \texttt{Vec}\ A\ n)\\
&(\underline{e_1} : \texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n)(\underline{e_2} : \texttt{cons}\ m\ x\ xs \equiv_{\texttt{Vec}\ A\ e_1} \texttt{cons}\ n\ y\ ys)\\
&\simeq (\underline{m}\ n : \mathbb{N})(x\ y : A)(xs : \texttt{Vec}\ A\ m)(ys : \texttt{Vec}\ A\ n)\\
&\quad (\underline{e_1} : m \equiv_{\mathbb{N}} n)(\underline{e_2} : x \equiv_A y)(\underline{e_3} : xs \equiv_{\texttt{Vec}\ A\ e_1} ys)\\
&\simeq (n : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ n)
\end{aligned}
\tag{30}
$$

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

To apply injectivity of cons*, the type of the equation has to be of the form* Vec *A e where e refers to a previous equation. This implies that this rule cannot be applied directly to an equation of the form* cons *n x xs* $\equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)}$ cons *n y ys where xs* : Vec *A n and ys* : Vec *A n have the same length 'on the nose'. We show how to solve this deficiency in Section 6.*

Example 40. *In the previous example, it was not really necessary to simplify the equation between the indices together with the equation between the constructors, as we could also have applied* $\texttt{injectivity}_{\texttt{suc}}$ *to the equation* suc *m* $\equiv_{\mathbb{N}}$ suc *n. However, sometimes this simplification gives a real increase to the power of unification. For example, let* $f : A \to B$ *be a (possibly very complex) function, then in general there is no way to solve an equation of the form* $f\ x \equiv_B f\ y$*. Now consider the following datatype:*

$$
\begin{aligned}
&\textbf{\textit{data}}\ \texttt{Im}\ f : B \to \texttt{Set}\ \textbf{\textit{where}}\\
&\quad \texttt{image} : (x : A) \to \texttt{Im}\ f\ (f\ x)
\end{aligned}
\tag{31}
$$

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

$$\begin{aligned}&(x\ y : A)(\underline{e_1} : f\ x \equiv_B f\ y)(\underline{e_2} : \texttt{image}\ x \equiv_{\texttt{Im}\ f\ e_1} \texttt{image}\ y)\\ &\quad\simeq (x\ \underline{y} : A)(\underline{e} : x \equiv_A y)\\ &\quad\simeq (x : A)\end{aligned} \tag{32}$$

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

Contrast this with the unification problem

$$(x\ y : A)(e_1 : \texttt{Im}\ f\ (f\ x) \equiv_{\texttt{Set}} \texttt{Im}\ f\ (f\ y))(e_2 : \texttt{image}\ x \equiv_{e_1} \texttt{image}\ y) \tag{33}$$

Here, it is not allowed to use injectivity on the second equation e_2 *since its type is not a datatype but a variable* e_1*. Like in Example 2, there is no way to distinguish between these two cases unless we keep track of the dependency of the type of* e_2 *on the equation* e_1*. Wrongly applying injectivity in situations like this led to the problems described by Abel (2015a,c) on the Agda bug tracker.*

Example 41. *Let* `D` $:$ `Bool` $\to$ `Set` *be an indexed datatype with two constructors* `tt` $:$ `D true` *and* `ff` $:$ `D false`*. Then the conflict rule between* `tt` *and* `ff` *gives us the following equivalence:*

$$(e_1 : \texttt{true} \equiv_{\texttt{Bool}} \texttt{false})(e_2 : \texttt{tt} \equiv_{\texttt{D}\ e_1} \texttt{ff}) \simeq \bot \tag{34}$$

On the other hand, the conflict rule cannot be applied if the first equation is between the types `D true` *and* `D false`*:*

$$(e_1 : \texttt{D true} \equiv_{\texttt{Set}} \texttt{D false})(e_2 : \texttt{tt} \equiv_{e_1} \texttt{ff}) \not\simeq \bot \tag{35}$$

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

Example 42. *This example is based on issue #1071 on the Agda bug tracker (Danielsson, 2014). Let* A $:$ `Set`*,* F $:$ `Set` $\to$ `Set` *and* `P` $:$ `Set` $\to$ `Set`$_1$ *be a datatype with one constructor* `c` $:$ $(A : \texttt{Set}) \to \texttt{P}\ (F\ A)$*. Then, we have*

$$\begin{aligned}&(f : F\ A)(R : \texttt{Set})(f' : F\ R)\\ &(e_1 : F\ A \equiv_{\texttt{Set}} F\ R)(e_2 : f \equiv_{e_1} f')(e_3 : \texttt{c}\ A \equiv_{\texttt{P}\ e_1} \texttt{c}\ R)\\ &\quad\simeq (f : F\ A)(R : \texttt{Set})(f' : F\ R)(e_3' : A \equiv_{\texttt{Set}} R)(e_2 : f \equiv_{F\ e_3'} f')\\ &\quad\simeq (f : F\ A)(f' : F\ A)(e_2 : f \equiv_{F\ A} f')\\ &\quad\simeq (f : F\ A)\end{aligned} \tag{36}$$

At each point during the unification process, there is only one valid way to proceed. At the first step, the second equation $f \equiv_{e_1} f'$ *cannot be solved right away as the type is*

heterogeneous and the solution rule only applies to homogeneous equations. The first equation cannot be solved either as this would require injectivity of the functor F. The only possibility is to apply the injectivity of c *to the third equation. At the second step,* $f \equiv_{F\, e'_3} f'$ *cannot be solved because the type* $F\ e'_3$ *is heterogeneous, so* e'_3 *has to be solved first instead.*

4.3 Rules for record types

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

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

Definition 43. *A* record type $\mathtt{R} : \mathtt{Set}_\ell$ *is defined by a number of* fields *(also called* projections*):*

$$\begin{aligned} \mathtt{f_1} &: (r : R) \to A_1 \\ \mathtt{f_2} &: (r : R) \to A_2\ (\mathtt{f_1}\ r) \\ &\vdots \\ \mathtt{f_n} &: (r : R) \to A_n\ (\mathtt{f_1}\ r)\ \ldots\ (\mathtt{f_{n-1}}\ r) \end{aligned} \tag{37}$$

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:*

$$\mathtt{f_i}\ (\mathtt{record}\{\mathtt{f_1} = x_1; \ldots; \mathtt{f_n} = x_n\}) = x_i \tag{38}$$

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

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

$$r = \mathtt{record}\{\mathtt{f_1} = \mathtt{f_1}\ r; \ldots; \mathtt{f_n} = \mathtt{f_n}\ r\} \tag{39}$$

We use the η-law to construct two unification rules. The first rule applies η to expand a variable of record type into its constituent fields, while the second rule performs a similar expansion on an equation between two elements of a record type.[4]

Lemma 44. *Let* $\mathtt{R} : \mathtt{Set}_\ell$ *be a record type with fields given by Equation* (37)*. Then we have an equivalence:*

$$\eta\mathtt{var_R} : (r : \mathtt{R}) \simeq (f_1 : A_1) \ldots (f_n : A_n\ f_1\ \ldots\ f_{n-1}) \tag{40}$$

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

Construction of $\eta\mathtt{var}_\mathtt{R}$. We define $\eta\mathtt{var}_\mathtt{R}\ r$ by $\mathtt{f}_1\ r;\ldots;\mathtt{f}_\mathtt{n}\ r$, and $\eta\mathtt{var}_\mathtt{R}^{-1}\ f_1\ \ldots\ f_n$ by $\mathtt{record}\{\mathtt{f}_1 = f_1;\ldots;\mathtt{f}_\mathtt{n} = f_n\}$. The proofs of both `isLinv` and `isRinv` is $\overline{\mathtt{refl}}$: in the former case this is type-correct because of the η-law (39), and in the latter case because of the computation rules for projections (38). □

Example 45. *This rule is especially useful for solving equations where one side is a projection applied to a variable. Consider the type* $A \times B = \Sigma_{\_:A} B$. *Then we can solve the equation* $\mathtt{fst}\ p \equiv_\mathbb{N} \mathtt{zero}$ *as follows:*

$$\begin{aligned}&(\underline{p} : \mathbb{N} \times \mathbb{N})(e : \mathtt{fst}\ p \equiv_\mathbb{N} \mathtt{zero})\\ &\quad\simeq (\underline{x} : \mathbb{N})(y : \mathbb{N})(\underline{e} : x \equiv_\mathbb{N} \mathtt{zero})\\ &\quad\simeq (y : \mathbb{N})\end{aligned} \tag{41}$$

Here the composite telescope mapping $\sigma : (y : \mathbb{N}) \to (p : \mathbb{N} \times \mathbb{N})(e : \mathtt{fst}\ p \equiv_\mathbb{N} \mathtt{zero})$ *from bottom to top is* $\lambda y.(\mathtt{zero}, y);\mathtt{refl}$.

Lemma 46. *Let* $\mathtt{R} : \mathtt{Set}_\ell$ *be a record type with fields given by Equation* (37). *Then we have an equivalence:*

$$\eta\mathtt{eq}_\mathtt{R} : (e : r \equiv_R s) \simeq (e_1 : \mathtt{f}_1\ r \equiv_{A_1} \mathtt{f}_1\ s)\ldots(e_n : \mathtt{f}_\mathtt{n}\ r \equiv_{A_n\ e_1\ \ldots\ e_{n-1}} \mathtt{f}_\mathtt{n}\ s) \tag{42}$$

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

Example 47. *This rule is useful when one side of an equation is of the form* $\mathtt{record}\{\ldots\}$. *If* $f : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$, *then we can solve the equation* $x, y \equiv_{\mathbb{N}\times\mathbb{N}} f\ z$ *as follows:*

$$\begin{aligned}&(x\ y\ z : \mathbb{N})(\underline{e} : x, y \equiv_{\mathbb{N}\times\mathbb{N}} f\ z)\\ &\quad\simeq (\underline{x}\ y\ z : \mathbb{N})(\underline{e_1} : x \equiv_\mathbb{N} \mathtt{fst}\ (f\ z))(e_2 : y \equiv_\mathbb{N} \mathtt{snd}\ (f\ z))\\ &\quad\simeq (\underline{y}\ z : \mathbb{N})(\underline{e_2} : y \equiv_\mathbb{N} \mathtt{snd}\ (f\ z))\\ &\quad\simeq (z : \mathbb{N})\end{aligned} \tag{43}$$

5 Computational behaviour of unification rules

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 is important for the applications we have in mind, in particular, the translation of pattern matching to eliminators (Cockx et al., 2016b).

In particular, an important step during the translation of a case split to the application of an eliminator is to generate auxiliary equations that are then solved by unification (Definition 27). However, in the end, these equations are filled in with $\overline{\mathtt{refl}}$.

Example 48. *Consider the definition of the function* `tail` *(Example 1):*

$$\begin{aligned}&\mathtt{tail} : (n : \mathbb{N}) \to \mathtt{Vec}\ A\ (\mathtt{suc}\ n) \to \mathtt{Vec}\ A\ n\\ &\mathtt{tail}\ .m\ (\mathtt{cons}\ m\ x\ xs) = xs\end{aligned} \tag{44}$$

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

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

- *In the first case, we get a negative unifier*

$$f_1 : (n : \mathbb{N})(\mathtt{zero} \equiv_{\mathbb{N}} \mathtt{suc}\ n) \simeq \bot \tag{45}$$

  and in particular $f\ n\ e : \bot$*, so the case can be handled by applying* $\mathtt{absurd} : (A : \mathtt{Set}_\ell) \to \bot \to A$.
- *In the second case, we get a positive unifier*

$$f_2 : (m\ n : \mathbb{N})(x : A)(xs : \mathtt{Vec}\ A\ m)(\mathtt{suc}\ m \equiv_{\mathbb{N}} \mathtt{suc}\ n) \simeq (m : \mathbb{N})(x : A)\ (xs : \mathtt{Vec}\ A\ m) \tag{46}$$

  This unifier is used in the further translation of the right-hand side of $\mathtt{tail}$.

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

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

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

Definition 49 (Strong unification rule). *A positive unification rule* $r : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$ *is a* strong unification rule *if for any* Γ_0 *and for any* $\bar{s}$ *and* $\bar{s}'$ *such that* $\Gamma_0 \vdash \overline{\mathtt{refl}} : \bar{u}[\Gamma \mapsto \bar{s}] \equiv_\Delta \bar{v}[\Gamma \mapsto \bar{s}]$ *and* $\Gamma_0 \vdash \overline{\mathtt{refl}} : \bar{u}'[\Gamma' \mapsto \bar{s}'] \equiv_{\Delta'} \bar{v}'[\Gamma' \mapsto \bar{s}']$*, it satisfies the following five properties:*

1. $\Gamma_0 \vdash r\ \bar{s}\ \overline{\mathtt{refl}} = (\bar{t}'; \overline{\mathtt{refl}}) : \Gamma'(\bar{e}' : \bar{u}' \equiv_{\Delta'} \bar{v}')$ *for some* $\bar{t}' : \Gamma'$.
2. $\Gamma_0 \vdash \mathtt{linv}\ r\ \bar{s}'\ \overline{\mathtt{refl}} = (\bar{t}_1; \overline{\mathtt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ *for some* $\bar{t}_1 : \Gamma$.
3. $\Gamma_0 \vdash \mathtt{rinv}\ r\ \bar{s}'\ \overline{\mathtt{refl}} = (\bar{t}_2; \overline{\mathtt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ *for some* $\bar{t}_2 : \Gamma$.
4. $\Gamma_0 \vdash \mathtt{isLinv}\ r\ \bar{s}\ \overline{\mathtt{refl}} = \overline{\mathtt{refl}} : \mathtt{linv}\ r\ (r\ \bar{s}\ \overline{\mathtt{refl}}) \equiv_{\Gamma(\bar{e}:\bar{u}\equiv_\Delta\bar{v})} (\bar{s}; \overline{\mathtt{refl}})$.
5. $\Gamma_0 \vdash \mathtt{isRinv}\ r\ \bar{s}'\ \overline{\mathtt{refl}} = \overline{\mathtt{refl}} : r\ (\mathtt{rinv}\ r\ \bar{s}'\ \overline{\mathtt{refl}}) \equiv_{\Gamma'(\bar{e}':\bar{u}'\equiv_{\Delta'}\bar{v}')} (\bar{s}'; \overline{\mathtt{refl}})$.

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

Lemma 50. *If* r *and* r' *are strong unification rules, then* $r \circ r'$ *is one as well.*

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

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

Lemma 51. `solution` *and* $\texttt{injectivity}_c$ *are strong unification rules.*

Proof. This follows directly from the construction of these rules (see Lemma 29 for `solution`, and Lemma 81 for $\texttt{injectivity}_c$.). □

For the deletion rule (30), the computational behaviour depends on the type of the equation being eliminated and the construction of the proof of K. In a theory with a general K rule, the deletion rule is also a strong unification rule. However, if we construct K for a specific datatype such as $\mathbb{N}$ from the basic eliminator, then the resulting unification rule will not be strong as evaluation gets stuck in case the left- and right-hand side of the equation are not of the form `zero` or `suc` m.

Lemma 52. $\eta\texttt{var}_R$ *and* $\eta\texttt{eq}_R$ *are strong unification rules.*

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

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

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

Definition 53 (Strong unifier). *A most general unifier* $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ *is* strong *if it satisfies the following four properties:*

1. $\Gamma' \vdash \texttt{linv}\ f\ \Gamma' = (\bar{t}_1; \overline{\texttt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ *for some* $\bar{t}_1$.
2. $\Gamma' \vdash \texttt{rinv}\ f\ \Gamma' = (\bar{t}_2; \overline{\texttt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ *for some* $\bar{t}_2$.
3. *For any* Γ_0 *and* $\bar{s}$ *such that* $\Gamma_0 \vdash \overline{\texttt{refl}} : \bar{u}[\Gamma \mapsto \bar{s}] \equiv_\Delta \bar{v}[\Gamma \mapsto \bar{s}]$, *we have* $\Gamma_0 \vdash \texttt{isLinv}\ f\ \bar{s}\ \overline{\texttt{refl}} = \overline{\texttt{refl}} : \texttt{linv}\ f\ (f\ \bar{s}\ \overline{\texttt{refl}}) \equiv_{\Gamma(\bar{e}:\bar{u}\equiv_\Delta\bar{v})} (\bar{s}; \overline{\texttt{refl}})$.
4. $\Gamma' \vdash \texttt{isRinv}\ f\ \Gamma' = \overline{\texttt{refl}} : f\ (\texttt{rinv}\ f\ \Gamma') \equiv_{\Gamma'} \Gamma'$.

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

Lemma 54. *If* $f = r_1 \circ r_2 \circ \ldots \circ r_n : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ *is composed of strong unification rules* $r_1, r_2, \ldots, r_n$, *then* f *is a strong unifier.*

Proof. Since a strong unifier is a special case of a strong unification rule, this follows directly from Lemma 50. □

Lemma 55. *If* $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ *is a strong unifier, then* $\Gamma' \vdash \texttt{linv}\ f\ \Gamma' = \texttt{rinv}\ f\ \Gamma' : \Gamma$.

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

$$\Gamma' \vdash \texttt{rinv}\ f\ \Gamma' = (\bar{s}_2; \overline{\texttt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{47}$$

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

$$\Gamma' \vdash \overline{\texttt{refl}} : \bar{u}[\Gamma \mapsto \bar{s}_2] \equiv_\Delta \bar{v}[\Gamma \mapsto \bar{s}_2] \tag{48}$$

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

$$\Gamma' \vdash \texttt{isLinv}\ f\ \bar{s}_2\ \overline{\texttt{refl}} = \overline{\texttt{refl}} : (\bar{s}_2; \overline{\texttt{refl}}) \equiv_{\Gamma(\bar{e}:\bar{u}\equiv_\Delta\bar{v})} (\bar{s}_2; \overline{\texttt{refl}}) \tag{49}$$

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

$$\Gamma' \vdash \texttt{linv}\ f\ (f\ \bar{s}_2\ \overline{\texttt{refl}}) = (\bar{s}_2; \overline{\texttt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{50}$$

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

$$\Gamma' \vdash f\ (\texttt{rinv}\ f\ \Gamma') = \Gamma' : \Gamma' \tag{51}$$

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

$$\Gamma' \vdash f\ \bar{s}_2\ \overline{\texttt{refl}} = \Gamma' : \Gamma' \tag{52}$$

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

$$\Gamma' \vdash \texttt{linv}\ f\ (f\ \bar{s}_2\ \overline{\texttt{refl}}) = \texttt{linv}\ f\ \Gamma' : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{53}$$

Putting this together with Equation (50) gives us that

$$\Gamma' \vdash \texttt{linv}\ f\ \Gamma' = (\bar{s}_2; \overline{\texttt{refl}}) : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \tag{54}$$

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

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

When applying specialization by unification to construct a function $m : (\bar{x} : \Gamma) \to (\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to T\ \bar{x}\ \bar{e}$ from the subgoal $m^s : (\bar{x}' : \Gamma') \to T\ (\texttt{linv}\ f\ \bar{x}')$, we expect m to have 'the same' computational behaviour as m^s in case the equations $\bar{u} \equiv_\Delta \bar{v}$ are actually satisfied. This is the content of the following lemma.

Lemma 56. *If* f *is a strong unifier (Definition 53), then the function* m *constructed through specialization by unification (Definition 27) satisfies the definitional equality* $m\ (f^{-1}\ \bar{x}') = m^s\ \bar{x}'$ *for any* $\bar{x}' : \Gamma'$.

Proof. Remember that for a strong unifier f, $\texttt{linv}\ f$ and $\texttt{rinv}\ f$ are definitionally equal, so it is fine to write f^{-1} here. By the first property of a strong unifier, we have $f^{-1}\ \bar{x}' = \bar{s}; \overline{\texttt{refl}}$ for some $\bar{s} : \Gamma$. By the third property, this implies that

$\texttt{isLinv}\ f\ (f^{-1}\ \bar{x}') = \overline{\texttt{refl}}$. By the fourth property, we also have $f\ (f^{-1}\ \bar{x}') = \bar{x}'$. So we have

$$\begin{aligned} m\ (f^{-1}\ \bar{x}') &= \overline{\texttt{subst}}\ T\ (\texttt{isLinv}\ f\ (f^{-1}\ \bar{x}'))\ (m^s\ (f\ (f^{-1}\ \bar{x}'))) \\ &= \overline{\texttt{subst}}\ T\ \overline{\texttt{refl}}\ (m^s\ \bar{x}') \\ &= m^s\ \bar{x}' \end{aligned} \tag{55}$$

□

The fact that $m\ (\bar{t}\ \overline{\texttt{refl}})$ evaluates to $m^s\ (f\ \bar{t}\ \overline{\texttt{refl}})$ ensures that the computational behaviour corresponds to the clause written by the user in the translation of pattern matching to eliminators.

Discussion about the definition of a strong unifier. There are other possible definitions of a strong unifier. In particular, to guarantee the good computational properties of functions constructed through specialization by unification (Lemma 56), we only need properties 1, 3, and a weaker version of property 4. In our previous work (Cockx *et al.*, 2016a), we used an even weaker definition of a strong unifier:

Definition 57 (DEPRECATED, version from Cockx *et al.* (2016a)). *A most general unifier* $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$ *is* strong *if for any* $\bar{x}' : \Gamma'$*, it satisfies the following two properties:*

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

This definition ensures exactly the properties needed for Lemma 56 to hold. However, we require another property in order to preserve clauses as definitional equalities in the translation of dependent pattern matching to eliminators (Cockx, 2017), namely that $f^{-1}\ \bar{x}'$ must be definitionally equal to something of the form $\bar{s}; \overline{\texttt{refl}}$. Additionally, in various places, we relied implicitly on the fact that $\texttt{linv}\ f$ and $\texttt{rinv}\ f$ should be definitionally equal. These discoveries lead to our current definition of a strong unifier.

6 Higher dimensional unification

When constructing the indexed versions of the injectivity, conflict, and cycle rules (Section 4.2), we required that the telescope of the equations on the left-hand side should be exactly $\overline{\texttt{D}} = (\bar{u} : \Xi)(x : \texttt{D}\ \bar{u})$. This means these rules can only be applied to an equation where the type is *fully general*, i.e. a datatype applied to distinct equality proofs for its indices. This is convenient when the equations we start with are of this form because it allows us to simplify all equations at the same time.

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

Example 58. *Suppose the unification algorithm is trying to solve an equation*

$$(e : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)} \texttt{cons}\ n\ y\ ys) \tag{56}$$

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

There is no fundamental reason why unification should fail on this example. On the other hand, always applying the injectivity rule even when the indices are not fully general is unsound (Example 4). This is not just a theoretical problem either: see, for example, issues #1411 and #1775 on the Agda bug tracker (Abel, 2015b; Sicard-Ramírez, 2016).

In the previous work, we tried different approaches to solve this problem that worked in some cases but were ultimately unsatisfactory. In Cockx *et al.* (2014), we restricted all the unification rules to the homogeneous equations and additionally imposed a *self-unifiability criterion* to the indices of the datatype when applying the injectivity rule. In practice, this meant that the injectivity rule could only be applied when the indices consisted of the closed constructor forms only (e.g. `suc` (`suc zero`), but not `suc` n), a severe restriction to the applicability of the rule. In Cockx *et al.* (2016a), we used the general (heterogeneous) version of the injectivity rule and relied on the *reverse unification* to generalize the indices. This method had some potential in theory, but turned out to be too difficult to implement in practice. Neither did we take into account the type of the constructor in question, so we were unable to include useful heuristics such as forced constructor arguments (Brady, McBride, and McKinna, 2004).

In this section, we describe a general technique for solving equations between constructors of indexed datatypes. First, we study why the problem is so difficult by looking at the analogous problem for the conflict and cycle rules, and make a first attempt at generalizing the injectivity rule (Section 6.1). We continue to show how to generalize the equality proofs in the indices in the general case by introducing new equations between equality proofs (Section 6.2). Borrowing terminology from homotopy type theory, we call them *higher dimensional equations*. To solve these higher dimensional equations, we show how to lift existing unification rules to higher dimensions (Section 6.3).

6.1 Generalizing unification rules

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

Lemma 59. *Consider a unification problem of the form* $\bar{s}_1 ; \mathtt{c}_1\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2 ; \mathtt{c}_2\ \bar{t}_2$, *where* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ *is a datatype and* $\mathtt{c}_1 : \Delta_1 \to \mathtt{D}\ \bar{u}_1$ *and* $\mathtt{c}_2 : \Delta_2 \to \mathtt{D}\ \bar{u}_2$ *are two distinct constructors of* $\mathtt{D}$. *Then we have an equivalence*

$$\mathtt{conflict}'_{\mathtt{c}_1,\mathtt{c}_2} : \left(\bar{s}_1 ; \mathtt{c}_1\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2 ; \mathtt{c}_2\ \bar{t}_2\right) \simeq \bot \tag{57}$$

The indices $\bar{v}$ are arbitrary, i.e. they do not have to be variables like in the standard conflict rule (Lemma 37). However, the type of the final equation still has to be the datatype `D` applied to these indices. In particular, it cannot be a variable itself, or else we would run into the problem described in Example 41.

Before we give the proof of this lemma, we first want to show how a naive proof attempt fails. It goes as follows: to construct a function $(\bar{s}_1 ; \mathtt{c}_1\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})}\ \bar{s}_2 ; \mathtt{c}_2\ \bar{t}_2) \to \bot$, it suffices (by the $\overline{\mathsf{J}}$ rule) to construct a function $\mathtt{c}_1\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}}\ \mathtt{c}_2\ \bar{t}_2 \to \bot$. This function is constructed by calling the indexed conflict rule (37) with $\overline{\mathtt{refl}}$ for the proof of $\bar{u}_1[\Delta_1 \mapsto \bar{t}_1] \equiv_\Xi \bar{u}_2[\Delta_2 \mapsto \bar{t}_2]$. Since any function to $\bot$ is an equivalence, we are done.

Think a moment about what is wrong with this proof. It uses the $\overline{\mathsf{J}}$ rule to eliminate the equations $\bar{s}_1 \equiv_\Phi \bar{s}_2$, but there is no guarantee that $\bar{s}_1$ or $\bar{s}_2$ are in fact variables. Moreover, their structure as a term may be important for satisfying the assumptions of the lemma, so simply generalizing the statement of the lemma is not possible. In other words, the error in this proof attempt stems from a confusion about the status of $\bar{s}_1$ and $\bar{s}_2$ as variables at the metalevel, while they can be arbitrary terms at the object level!

We work around this issue by using the following lemma:

Lemma 60. *Let $f : A \to B$, $P : B \to \mathtt{Set}$, $e : s \equiv_A t$, $u : P\ (f\ s)$, and $v : P\ (f\ t)$. Then the types $u \equiv_{P\ (f\ e)} v$ and $u \equiv_{P\ (\mathtt{cong}\ f\ e)} v$ are equivalent.*

Proof. Notice the rather subtle difference between these two types: the first one expands to

$$\mathtt{subst}\ (P \circ f)\ e\ u \equiv_{P\ t} v \tag{58}$$

while the second one expands to

$$\mathtt{subst}\ P\ (\mathtt{cong}\ f\ e)\ u \equiv_{P\ t} v \tag{59}$$

To prove that they are equivalent, it is sufficient to prove that $\mathtt{subst}\ (P \circ f)\ e\ u \equiv_{P\ t} \mathtt{subst}\ P\ (\mathtt{cong}\ f\ e)\ u$. But this follows directly by eliminating e using J. □

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

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}_1\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}[\Phi \mapsto \bar{e}_1]}\ \mathtt{c}_2\ \bar{t}_2) \tag{60}$$

By Lemma 60, the type of e_2 is equivalent to $\mathtt{c}_1\ \bar{t}_1 \equiv_{\mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\Phi.\,\bar{v})\ \bar{e})}\ \mathtt{c}_2\ \bar{t}_2$. So we call the conflict rule (37) with arguments $(\overline{\mathtt{cong}}\ (\lambda\Phi.\,\bar{v})\ \bar{e}_1); e_2$ to get an element of type $\bot$. Since any function to $\bot$ is an equivalence, this finishes the proof. □

Similarly, we can generalize the cycle rule:

Lemma 61. *Consider a unification problem of the form $\bar{s}_1 ; t_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})}\ \bar{s}_2 ; t_2$, where $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ is a datatype and $t_1 \prec t_2$. Then we have an equivalence*

$$\mathtt{cycle}'_{t_1,t_2} : \left(\bar{s}_1 ; t_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})}\ \bar{s}_2 ; t_2\right) \simeq \bot \tag{61}$$

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

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

Lemma 62 (Generalized injectivity). *Consider a unification problem of the form* $\bar{s}_1 ; \mathtt{c}\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2 ; \mathtt{c}\ \bar{t}_2$ *where* $\mathtt{D} : \Xi \to \mathtt{Set}_\ell$ *is a datatype with (at least) one constructor* $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$*, and assume* Ξ *satisfies UIP, i.e. we have* $\mathtt{deletion}_{\bar{x}} : (\bar{e} : \bar{x} \equiv_\Xi \bar{x}) \simeq ()$ *for all* $\bar{x} : \Xi$*. Then we have an equivalence*

$$\mathtt{injectivity'_c} : \left(\bar{s}_1 ; \mathtt{c}\ \bar{t}_1 \equiv_{\Phi(x:\mathtt{D}\ \bar{v})} \bar{s}_2 ; \mathtt{c}\ \bar{t}_2\right) \simeq (\bar{s}_1 ; \bar{t}_1 \equiv_{\Phi\Delta} \bar{s}_2 ; \bar{t}_2) \tag{62}$$

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

Construction of $\mathtt{injectivity'_c}$. As for the previous lemma, we expand the definition of telescopic equality and apply Lemma 60 to get to

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\Phi.\,\bar{v})\ \bar{e}_1)}\ \mathtt{c}\ \bar{t}_2) \tag{63}$$

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

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_1 : \bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2])\\ &(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ (\overline{\mathtt{cong}}\ (\lambda\Phi.\,\bar{v})\ \bar{e}_1)}\ \mathtt{c}\ \bar{t}_2)\end{aligned} \tag{64}$$

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

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_1 : \bar{v}[\Phi \mapsto \bar{s}_1] \equiv_\Xi \bar{v}[\Phi \mapsto \bar{s}_2])(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{e}'_1}\ \mathtt{c}\ \bar{t}_2) \tag{65}$$

Finally, we can apply the injectivity rule (36) to prove that the part of the telescope containing $\bar{e}'_1$ and e_2 is equivalent to $\bar{t}_1 \equiv_\Delta \bar{t}_2$, so the previous telescope is equivalent to

$$(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}_2 : \bar{t}_1 \equiv_\Delta \bar{t}_2) \tag{66}$$

which is what we wanted to prove. □

Like for the deletion rule, this generalized injectivity rule usually will not be a strong rule because its computational behaviour depends on the construction of the proof UIP for the index types.

6.2 A generalized injectivity rule

The generalized injectivity rule from the previous section is unsatisfactory because it requires the index types of the datatype to satisfy UIP. This means we did not actually solve the problem of depending on UIP yet, we only moved it to the indices. However, the proof taught us something about how to solve the problem in general: it introduced new equality proofs $\bar{e}'_1$ and used UIP to substitute these for the indices of D, allowing us to apply the injectivity rule. In other words, it moved the problem from talking about equalities between *terms* to equalities between *equality proofs*.

In this section, we show how to apply this idea in a more general way to remove the dependency on UIP completely. We do this by applying—what else—unification to the equations between the indices of the datatype. Since the indices in the type of

an equation can depend on the equality proofs of the previous equations, this means we have to solve not just the equalities between the terms but also the equalities between other equality proofs, i.e. higher dimensional equations.

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

In general, whenever the unification algorithm encounters a higher dimensional unification problem $\bar{u} \equiv_{\bar{x} \equiv_\Delta \bar{y}} \bar{v}$, it lowers it by one dimension to the problem $\bar{u} \equiv_\Delta \bar{v}$ where the equation variables in $\bar{u}$ and $\bar{v}$ are treated as regular variables. If it manages to find a solution to this one-dimensional problem, it can then *lift* this solution to get a solution to the original problem. The technical result that makes this possible is Lemma 72 in the next section.

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

Example 63. *Consider the unification problem:*

$$\Gamma(e : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)} \texttt{cons}\ n\ y\ ys) \tag{67}$$

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

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

$$\begin{aligned} &\Gamma(e : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)} \texttt{cons}\ n\ y\ ys) \\ &\quad \simeq \Gamma(e_1 : \texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n)(e_2 : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ e_1} \texttt{cons}\ n\ y\ ys) \\ &\qquad (p : e_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl}) \end{aligned} \tag{68}$$

This is nothing but an application of the $\texttt{solution}$ *rule in the reverse direction, as applying* $\texttt{solution}$ *to* p *would bring us back to the first equation.*[5]

[5] This is the exact same technique as used by McBride (1998b): To do a case split on a variable $x : \texttt{Vec}\ A\ m$, where m is not fully general, he introduces a new variable $n : \mathbb{N}$ together with an equality $e : m \equiv_{\mathbb{N}} n$. This means that now $x : \texttt{Vec}\ A\ m$ where m are just variables, so it is possible to perform a case split on x. The only difference in our case is that we are working one dimension higher, i.e. we work with equations between the elements of the datatype instead of the elements of the datatype itself.

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

$$\begin{aligned}&\Gamma(\underline{e_1} : \texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n)(\underline{e_2} : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ e_1} \texttt{cons}\ n\ y\ ys)\\&(p : e_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl})\\&\quad\simeq \Gamma(e_1' : n \equiv_{\mathbb{N}} n)(e_2' : x \equiv_A y)(e_3' : xs \equiv_{\texttt{Vec}\ A\ e_1'} ys)\\&\quad\ (p : \texttt{cong}\ \texttt{suc}\ e_1' \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl})\end{aligned} \tag{69}$$

Applying the injectivity rule to e_2 has instantiated the variable e_2 with $\texttt{cong}\ \texttt{suc}\ e_1'$. *This instantiation is determined by the computational behaviour of* $\texttt{injectivity}_{\texttt{cons}}$ *(Lemma 31). As you can see, p is a non-trivial equation between equality proofs, i.e. a* higher dimensional equation.

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

$$(w_1' : \mathbb{N})(w_2' : A)(w_3' : \texttt{Vec}\ A\ w_1')(p : \texttt{suc}\ w_1' \equiv_{\mathbb{N}} \texttt{suc}\ n) \tag{70}$$

The equality proofs e_1', e_2', and e_3' from Equation (69) *have been replaced by regular variables w_1', w_2', and w_3'. To reflect this change,* $\texttt{cong}\ \texttt{suc}\ e_1' : \texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n$ *has been replaced by* $\texttt{suc}\ w_1'$ *and* $\texttt{refl} : \texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n$ *by* $\texttt{suc}\ n$.
Now this is a problem we know how to solve: We apply $\texttt{injectivity}_{\texttt{suc}}$ *and* $\texttt{solution}$ *to find an equivalence f between this telescope and $(w_2' : A)(w_3' : \texttt{Vec}\ A\ n)$. This solves the one-dimensional problem.*

Step 3: Lifting unifiers to a higher dimension. *How does this help us with the higher dimensional problem? By Lemma 72 (Section 6.3), we can* lift *the equivalence f to get a new equivalence $f^{\uparrow}$:*

$$\begin{aligned}&(e_1' : n \equiv_{\mathbb{N}} n)(e_2' : x \equiv_A y)(e_3' : xs \equiv_{\texttt{Vec}\ A\ e_1'} ys)\\&(p : \texttt{cong}\ \texttt{suc}\ e_1' \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl})\\&\quad\simeq (e_2'' : x \equiv_A y)(e_3'' : xs \equiv_{\texttt{Vec}\ A\ n} ys)\end{aligned} \tag{71}$$

This solves the higher dimensional equation p, as well as the reflexive equation e_1', without relying on the fact that $\mathbb{N}$ satisfies UIP!

Finally, we apply the $\texttt{solution}$ *rule twice to solve the equations e_2'' and e_3''. So putting everything together, we have found an equivalence between the original telescope* (67) *and $(n : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ n)$, solving the unification problem.*

Now that we have seen how to solve the problem in an example, let us try to generalize the solution. The main result of this section is the following theorem.

Theorem 64. *Let* $\texttt{D} : \Xi \to \texttt{Set}_\ell$ *be a datatype and* $\texttt{c} : \Delta \to \texttt{D}\ \bar{u}$ *be a constructor of* $\texttt{D}$. *Consider a unification problem of the form*

$$(\bar{e} : \bar{s}_1 ; \texttt{c}\ \bar{t}_1 \equiv_{\Phi(z : \texttt{D}\ \bar{v})} \bar{s}_2 ; \texttt{c}\ \bar{t}_2) \tag{72}$$

Suppose we have an equivalence $f : \Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v}) \simeq \Delta'$ Then we also have an equivalence $\texttt{injectivity}''^{f}_{\texttt{c}}$ *of type*

$$(\bar{e} : \bar{s}_1 ; \texttt{c}\ \bar{t}_1 \equiv_{\Phi(z : \texttt{D}\ \bar{v})} \bar{s}_2 ; \texttt{c}\ \bar{t}_2) \simeq (\bar{e}' : f\ \bar{s}_1\ \bar{t}_1\ \overline{\texttt{refl}} \equiv_{\Delta'} f\ \bar{s}_2\ \bar{t}_2\ \overline{\texttt{refl}}) \tag{73}$$

Moreover, if f is a strong unification rule, then so is this new equivalence.

The computational behaviour of the unifier f suddenly becomes relevant for the type of the resulting unification problem! In particular, we need the behaviour of f applied to $\overline{\texttt{refl}}$ to calculate the left- and right-hand sides of the new equations $\bar{e}'$.

Construction. We follow the same three steps as in Example 63, so if something is unclear it may help to take a look at the corresponding step in the example.

Step 1: generalizing the indices. First, we unfold the telescopic equality in Equation (72) and apply Lemma 60 to get an equivalence with $(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}_e}\ \mathtt{c}\ \bar{t}_2)$ where $\bar{v}_e = \overline{\texttt{cong}}\ (\lambda\Phi.\ \bar{v})\ \bar{e}_1$. The equality proofs $\bar{v}_e$ have type $\bar{u}_1 \equiv_\Xi \bar{u}_2$ where $\bar{u}_1$ and $\bar{u}_2$ stand for $\bar{u}[\Delta \mapsto \bar{t}_1]$ and $\bar{u}[\Delta \mapsto \bar{t}_2]$, respectively. To generalize $\bar{v}_e$, we introduce new variables $\bar{\imath} : \bar{u}_1 \equiv_\Xi \bar{u}_2$ together with equalities $\bar{p} : \bar{\imath} \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e$:

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}_e}\ \mathtt{c}\ \bar{t}_2)\\ &\quad\simeq (\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{\imath} : \bar{u}_1 \equiv_\Xi \bar{u}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{\imath}}\ \mathtt{c}\ \bar{t}_2)\\ &\qquad(\bar{p} : \bar{\imath} \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\end{aligned} \tag{74}$$

Since $\bar{\imath}$ consists of distinct equation variables, it is now possible to apply $\texttt{injectivity}_\mathtt{c}$ to the equation e_2. This gives us an equivalence:

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{\imath} : \bar{u}_1 \equiv_\Xi \bar{u}_2)(\underline{e_2} : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{\imath}}\ \mathtt{c}\ \bar{t}_2)(\bar{p} : \bar{\imath} \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\\ &\quad\simeq (\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_2 : \bar{t}_1 \equiv_\Delta \bar{t}_2)(\bar{p} : \bar{u}_e \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\end{aligned} \tag{75}$$

where $\bar{u}_e = \overline{\texttt{cong}}\ (\lambda\Delta.\ \bar{u})\ \bar{e}'_2$.

Step 2: Lowering the dimension of equations. Consider the one-dimensional version of this unification problem $\Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$, where the equality proofs $\bar{u}_e$ and $\bar{v}_e$ have been replaced by their lower dimensional variants $\bar{u}$ and $\bar{v}$, respectively. Since this is a one-dimensional unification problem, we can apply the known unification rules from Figure 8 to solve it. By assumption of the theorem, unification succeeds positively with most general unifier f as a result.

Step 3: Lifting unifiers to a higher dimension. Now we have to lift this solution back to the higher dimensional problem. This lifting is explained in the next subsection. Lemma 72 gives us a lifted equivalence $f^\uparrow$:

$$\begin{aligned}&(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(\bar{e}'_2 : \bar{t}_1 \equiv_\Delta \bar{t}_2)(\bar{p} : \bar{u}_e \equiv_{\bar{u}_1 \equiv_\Xi \bar{u}_2} \bar{v}_e)\\ &\quad\simeq (\bar{e}' : f\ \bar{s}_1\ \bar{t}_1\ \overline{\texttt{refl}} \equiv_{\Delta'} f\ \bar{s}_2\ \bar{t}_2\ \overline{\texttt{refl}})\end{aligned} \tag{76}$$

This is exactly what we need to solve the problem in Equation (75).

Now we combine the equivalences in Equations (74)–(76) to get the final equivalence (73).

To see why this is a strong unification rule, note that it is the composition of four equivalences: Lemma 60, $\texttt{solution}^{-1}$, $\texttt{injectivity}_\mathtt{c}$, and $f^\uparrow$. By Lemma 50, it is sufficient to prove that these four equivalences are the strong unification rules individually. The first two are strong by construction, and $\texttt{injectivity}_\mathtt{c}$ is a strong unification rule by Lemma 51. Finally, $f^\uparrow$ is strong too by Lemma 73. □

This finishes the application of higher dimensional unification to the equation e. We have solved the injectivity problem e, and there are no more higher dimensional

unification problems in the resulting equations $\bar{e}'_1$, so we can continue unification on the new problem as normal.

Having given the rule for higher dimensional unification, the presentation of our full unification algorithm is finished. The rules are summarized in Figure 9, including η-rules for record types (Lemmas 44 Lemma 46), generalized conflict and cycle (Lemmas 59 Lemma 61) and higher dimensional unification (Theorem 64).

It is impossible for higher dimensional unification to end in a negative success, as this would mean we are trying to solve an ill-typed equation. For example, we can never encounter a higher dimensional conflict:

$$\overline{\texttt{cong}}\ \texttt{c}_1\ \bar{e}_1 \equiv_{???} \overline{\texttt{cong}}\ \texttt{c}_2\ \bar{u}'\ \bar{v}'\ \bar{e}_2 \tag{77}$$

because the left-hand side has a type of the form $\texttt{c}_1\ \bar{u} \equiv \texttt{c}_1\ \bar{v}$ while the right-hand side has type $\texttt{c}_2\ \bar{u}' \equiv \texttt{c}_2\ \bar{v}'$. Likewise, a higher dimensional cycle would be

$$e \equiv_{???} \overline{\texttt{cong}}\ \texttt{c}\ e \tag{78}$$

where the left-hand side has some type $u \equiv v$, but the right-hand side has type $\texttt{c}\ \bar{u}' \equiv \texttt{c}\ \bar{v}'$, where u and v occur in $\bar{u}'$ and $\bar{v}'$, respectively.

To see how higher dimensional unification can be applied in a concrete situation, we now show a more limited but still very useful corollary of Theorem 64. In particular, this corollary formally justifies the notion of a *forced constructor argument* (Brady et al., 2004).

Definition 65 (Forced constructor argument). *A variable x occurs* rigidly *in a term t if either $t = x$ or t is of the form $\texttt{c}\ t_1\ \ldots\ t_n$, where x occurs rigidly in one of the t_i and t_i is not a forced argument of $\texttt{c}$.*

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

As an example, the argument $(n : \mathbb{N})$ is a forced argument of $\texttt{cons} : (n : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ n) \to \texttt{Vec}\ A\ (\texttt{suc}\ n)$. The concept of a forced constructor argument was introduced by Brady et al. (2004) for efficiently compiling dependently typed programs. We use it here to describe when it is safe to apply the **injectivity** rule.

Definition 66 (Invertible constructor). *A constructor $\texttt{c} : \Delta \to \texttt{D}\ \bar{u}$ is* invertible *if the indices $\bar{u}$ consist only of invertible constructors and variables bound in Δ, and no variable from Δ occurs more than once in a non-forced position in $\bar{u}$.*

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

Now we can justify why forced constructor arguments (of invertible constructors) can be skipped during unification.

Corollary 67. *Let $\texttt{c} : \Delta \to \texttt{D}\ \bar{u}$ be an invertible constructor (Definition 66) of the datatype $\texttt{D} : \Xi \to \texttt{Set}_\ell$ and $\bar{t}_1, \bar{t}_2 : \Delta$. Then we have an equivalence*

$$(e : \texttt{c}\ \bar{t}_1 \equiv_{\texttt{D}\ \bar{v}} \texttt{c}\ \bar{t}_2) \simeq (\bar{e}' : \bar{t}_1|_{\Delta'} \equiv_{\Delta'} \bar{t}_2|_{\Delta'}) \tag{79}$$

where $\bar{v} = \bar{u}[\Delta \mapsto \bar{t}_1] = \bar{u}[\Delta \mapsto \bar{t}_2]$, Δ' *is the telescope of non-forced arguments of* c *with the forced arguments filled in with the corresponding values from* $\bar{t}_1$, *and* $\bar{t}|_{\Delta'}$ *is the sublist of* $\bar{t}$ *corresponding to the variables occurring in* Δ'.

From the well-formedness of the type $\mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}} \mathtt{c}\ \bar{t}_2$, it follows that the forced arguments of $\bar{t}_1$ and $\bar{t}_2$ are equal, so it does not matter from which side we take them.

Construction. By definition of an invertible constructor, applying unification to the unification problem $\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$ ends in a positive success with result $f : \Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v}) \simeq \Delta'$ where the computational behaviour of f is to select the non-forced arguments of c from Δ. Applying Theorem 64 to f gives us the desired equivalence. □

In particular, when applying **injectivity** to an equation $\mathtt{c}\ \bar{s} = \mathtt{c}\ \bar{t}$, this corollary tells us that it is safe to skip unification of the forced arguments of c. This allows us to avoid some situations where unification would otherwise require the **deletion** rule.

Example 68. *Let* $\mathtt{Fin} : \mathbb{N} \to \mathtt{Set}$ *be the following datatype:*

$$\begin{aligned} &\textbf{\textit{data}}\ \mathtt{Fin} : \mathbb{N} \to \mathtt{Set}\ \textbf{\textit{where}} \\ &\quad \mathtt{fzero} : (n : \mathbb{N}) \to \mathtt{Fin}\ (\mathtt{suc}\ n) \\ &\quad \mathtt{fsuc}\ \ : (n : \mathbb{N}) \to \mathtt{Fin}\ n \to \mathtt{Fin}\ (\mathtt{suc}\ n) \end{aligned} \tag{80}$$

We apply Corollary 67 to solve the equation $\mathtt{fsuc}\ n\ x \equiv_{\mathtt{Fin}\ (\mathtt{suc}\ n)} \mathtt{fsuc}\ n\ y$. *Since the first argument of the constructor* fsuc *is forced, the corresponding equation* $n = n$ *can be skipped. So we get an equivalence of type* $\mathtt{fsuc}\ n\ x \equiv_{\mathtt{Fin}\ (\mathtt{suc}\ n)} \mathtt{fsuc}\ n\ y \simeq x \equiv_{\mathtt{Fin}\ n} y$ *In particular, we do not need to solve the (forced) equation* $n \equiv_{\mathbb{N}} n$.

6.3 Lifting unifiers to higher dimensions

We have seen how to apply higher dimensional unification to make the injectivity rule more generally applicable. In this section, we dive into the heart of the problem. Our core result that makes higher dimensional unification work is Lemma 72, telling us exactly how to update the left- and right-hand sides of the equations when lifting a unifier.

Suppose we have a unifier that we want to lift to a higher dimension. As a first attempt, we try to apply the following theorem from The Univalent Foundations Program (2013):

Theorem 69. *If a function* $f : A \to B$ *is an equivalence and* $x, y : A$, *then* $\mathtt{cong}\ f : x \equiv_A y \to f\ x \equiv_B f\ y$ *is also an equivalence.*

Construction. This is Theorem 2.11.1 from The Univalent Foundations Program (2013). □

Applying this theorem to a unifier $f : \Gamma(\bar{p} : \bar{a} \equiv_\Delta \bar{b}) \simeq \Gamma'$ results in an equivalence $\mathtt{cong}\ f : (\bar{e} : \bar{u}; \bar{r} \equiv_{\Gamma(\bar{p}:\bar{a}\equiv_\Delta\bar{b})} \bar{v}; \bar{s}) \simeq (\bar{e}' : f\ \bar{u}\ \bar{r} \equiv_{\Gamma'} f\ \bar{v}\ \bar{s})$, or expanding the definition of telescopic equality:

$$\mathtt{cong}\ f : (\bar{e} : \bar{u} \equiv_\Gamma \bar{v})(\bar{q} : \bar{r} \equiv_{\bar{a}_e \equiv_{\Delta_e} \bar{b}_e} \bar{s}) \simeq (\bar{e}' : f\ \bar{u}\ \bar{r} \equiv_{\Gamma'} f\ \bar{v}\ \bar{s}) \tag{81}$$

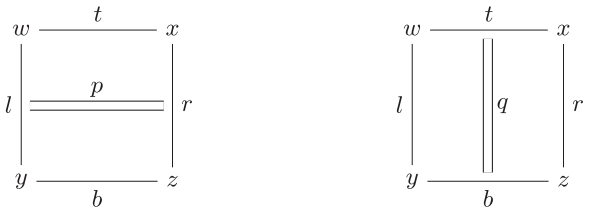

Fig. 10. The `Square` type represents the possible ways to fill a square defined by four equality proofs. (a) Horizontal filling, (b) vertical filling.

where $\bar{u}, \bar{v} : \Gamma$, $\bar{r} : \bar{a}_u \equiv_{\Delta_u} \bar{b}_u$, $\bar{s} : \bar{a}_v \equiv_{\Delta_v} \bar{b}_v$, and $\cdot_x$ is shorthand for $\cdot[\Gamma \mapsto \bar{x}]$. This is already *almost* what we need for higher dimensional unification, but not quite.

To better visualize the problem, we make use of the concept of a *square*, also called a *2-path* by The Univalent Foundations Program (2013):

Definition 70 (Square). *Let* $A : \mathtt{Set}_\ell$, $w, x, y, z : A$, $t : w \equiv_A x$, $b : y \equiv_A z$, $l : w \equiv_A y$, *and* $r : x \equiv_A z$. *The* square type `Square` $t\ b\ l\ r$ *is defined to be the dependent equality type* $l \equiv_{t \equiv_A b} r$.

The type $l \equiv_{t \equiv_A b} r$ can be written a little more explicitly as $l \equiv^{t;b}_{\_ \equiv_A \_} r$, or even more explicitly as $\overline{\mathtt{subst}}\ (\_ \equiv_A \_)\ (t;b)\ l \equiv_{x \equiv_A z} r$. If we imagine a square with top side t, bottom side b, left side l, and right side r, then `Square` $t\ b\ l\ r$ can be thought of as the type of identity proofs that fill this square horizontally as visualized in Figure 10(a).

There is a second way to construct a square type from four given points $w, x, y, z : A$ and equality proofs $t : w \equiv_A x$, $b : y \equiv_A z$, $l : w \equiv_A y$: We can 'flip' the square around its w–z axis, as illustrated by Figure 10(b). To get to our desired result, we need to rely on the fact that both square types are in fact equivalent:

Lemma 71 (Flipping squares). *Let* A : `Set`, $w, x, y, z : A$, $t : w \equiv_A x$, $b : y \equiv_A z$, $l : w \equiv_A y$, *and* $r : x \equiv_A z$. *Then we have an equivalence* `flip` $t\ b\ l\ r$: `Square` $t\ b\ l\ r \simeq$ `Square` $l\ r\ t\ b$.

Proof. The proof of this lemma consists completely of repeated applications of `J`. We start by constructing the function `flip` $t\ b\ l\ r$: `Square` $t\ b\ l\ r \to$ `Square` $l\ r\ t\ b$. First, by `J` on t and b, we can assume that $w = x$, $y = z$ and both t and b are `refl`, so we are left with the goal $l \equiv_{w \equiv_A y} r \to$ `refl` $\equiv_{l \equiv_A r}$ `refl`. The identity type in the function argument has become homogeneous, so we again apply `J`, giving us that $l = r$ and leaving us with the goal `refl` $\equiv_{l \equiv_a l}$ `refl`. Finally, one more application of `J` on $l : w \equiv_A y$ leaves us with the goal `refl` $\equiv_{w \equiv_A w}$ `refl`, which we solve with `refl`.

For the construction of the left and right inverse of `flip`, we just change the order of t, b, l, and r in the construction of `flip`. For the proofs that they are in fact inverses, the same sequence of applications of `J` as used in the construction of `flip` suffices. □

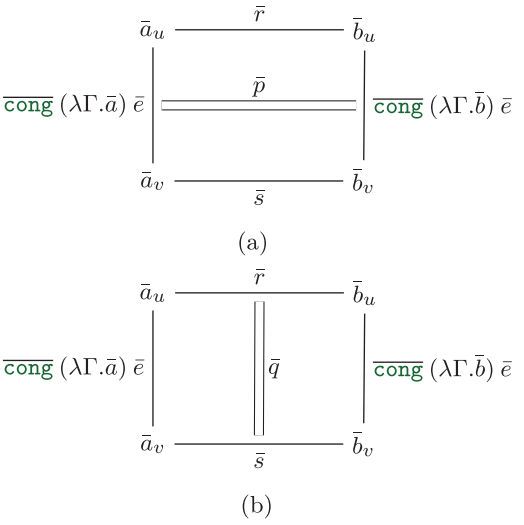

Fig. 11. To construct the equivalence in Lemma 72, we apply Lemma 71 to transform the horizontal filling $\bar{p}$ into a vertical one $\bar{q}$. (a) Horizontal filling $\bar{p}$, (b) vertical filling $\bar{q}$.

Now we prove the main lemma. When we applied this lemma in the last section, we only used it for $\bar{r} = \overline{\texttt{refl}}$ and $\bar{s} = \overline{\texttt{refl}}$, but the fully general version is not harder to prove so that is what we present here.

Lemma 72 (Lifting of unifiers). *Suppose we have a unifier* $f : \Gamma(\bar{p} : \bar{a} \equiv_\Delta \bar{b}) \simeq \Gamma'$ *and* $\bar{u}, \bar{v} : \Gamma$, $\bar{r} : \bar{a}_u \equiv_{\Delta_u} \bar{b}_u$, *and* $\bar{s} : \bar{a}_v \equiv_{\Delta_v} \bar{b}_v$.[6] *Then we have a* lifted unifier

$$\begin{aligned} f^\uparrow : (\bar{e} : \bar{u} \equiv_\Gamma \bar{v})(\bar{p} : \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{a})\ \bar{e} \equiv_{\bar{r} \equiv_{\Delta_e} \bar{s}} \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{b})\ \bar{e}) \\ \simeq (\bar{e}' : f\ \bar{u}\ \bar{r} \equiv_{\Gamma'} f\ \bar{v}\ \bar{s}) \end{aligned} \tag{82}$$

Construction. By Theorem 69, we already have the equivalence in Equation (81). By Lemma 60, the type of $\bar{q}$ is equivalent to $\bar{r} \equiv_{\overline{\texttt{cong}}\ (\lambda\Gamma.\bar{a})\ \bar{e} \equiv_{\Delta_e} \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{b})\ \bar{e}} \bar{s}$. If we think of this type as a square type, then Lemma 71 gives us that this type is equivalent to $\overline{\texttt{cong}}\ (\lambda\Gamma.\bar{a})\ \bar{e} \equiv_{\bar{r} \equiv_{\Delta_e} \bar{s}} \overline{\texttt{cong}}\ (\lambda\Gamma.\bar{b})\ \bar{e}$. This is illustrated in Figure 11. Composing this equivalence with $\texttt{cong}\ f$ gives us the desired equivalence $f^\uparrow$. □

Lemma 73. *If* $f : \Gamma(\bar{p} : \bar{a} \equiv_\Delta \bar{b}) \simeq \Gamma'$ *is a strong unifier, then so is* $f^\uparrow$.

Proof. $f^\uparrow$ is constructed as a composition of the equivalences $\texttt{cong}\ f$ (Theorem 69), Lemma 60, and $\texttt{flip}$ (Lemma 71). By Lemma 50, we just have to verify that each of these equivalences is a strong unification rule. But this can be verified by looking at their construction (in the case of $\texttt{cong}\ f$ also using the fact that f is a strong unifier). □

7 Translation to eliminators

In this section, we translate the injectivity, conflict, and cycle rules from Section 4 to the 'bare metal' of type theory: datatype eliminators. These eliminators encode

[6] We again write $\cdot_x$ for $\cdot[\Gamma \mapsto \bar{x}]$.

the basic induction principles associated to each datatype. By translating definitions by pattern matching to eliminators, we can be confident that they do not add any extra assumptions to the core theory.

For the rest of this section, let $\mathtt{D} : \Xi \to \mathtt{Set}_i$ be an inductive family (where Ξ is the telescope of the indices) with constructors $\mathtt{c}_1, \ldots, \mathtt{c}_k$. Without loss of generality, we assume that the non-recursive constructor arguments come before the recursive ones, so $\mathtt{c}_i$ has type

$$\mathtt{c}_i : \Delta_i \to (\Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1}) \to \ldots \to (\Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \mathtt{D}\ \bar{u}_i \tag{83}$$

We consider $\mathtt{D}$ to be already applied to its parameters, if it has any.

Definition 74. *The standard datatype eliminator* $\mathtt{elim}_\mathtt{D}$ *for* $\mathtt{D}$ *has type*

$$\begin{aligned} \mathtt{elim}_\mathtt{D}\ &: (P : \overline{\mathtt{D}} \to \mathtt{Set}_i)(m_1 : M_1) \ldots (m_k : M_k) \\ &\to (\bar{x} : \overline{\mathtt{D}}) \to P\ \bar{x} \end{aligned} \tag{84}$$

where the methods $m_1, \ldots, m_k$ *have type*

$$\begin{aligned} M_i = (\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \\ \to (h_1 : (\bar{s}_1 : \Phi_{i,1}) \to P\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \to \ldots \\ \to (h_{n_i} : (\bar{s}_{n_i} : \Phi_{i,n_i}) \to P\ \bar{v}_{i,n_i}\ (x_{n_i}\ \bar{s}_{n_i})) \\ \to P\ \bar{u}_i\ (\mathtt{c}_i\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \end{aligned} \tag{85}$$

The evaluation behaviour of the standard datatype eliminator is given by the following rule for $i = 1, \ldots, k$*:*

$$\begin{aligned} &\mathtt{elim}_\mathtt{D}\ P\ m_1\ \ldots\ m_k\ \bar{u}_i\ (\mathtt{c}_i\ \bar{t}\ x_1\ \ldots\ x_{n_i}) = \\ &\quad m_i\ \bar{t}\ x_1\ \ldots\ x_{n_i} \\ &\qquad (\lambda \bar{s}_1.\, \mathtt{elim}_\mathtt{D}\ P\ m_1\ \ldots\ m_k\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \\ &\qquad \ldots \\ &\qquad (\lambda \bar{s}_{n_i}.\, \mathtt{elim}_\mathtt{D}\ P\ m_1\ \ldots\ m_k\ \bar{v}_{i,n_i}(x_{n_i}\ \bar{s}_{n_i})) \end{aligned} \tag{86}$$

Example 75. *Following the pedagogy of McBride, Goguen, and McKinna (2006), we take binary trees as our example datatype. This type* $\mathtt{Tree}$ *is defined by the two constructors* $\mathtt{leaf} : \mathtt{Tree}$ *and* $\mathtt{node} : \mathtt{Tree} \to \mathtt{Tree} \to \mathtt{Tree}$*. The eliminator for* $\mathtt{Tree}$ *is*

$$\begin{aligned} \mathtt{elim}_\mathtt{Tree}\ &: (P : \mathtt{Tree} \to \mathtt{Set}_i)(m_\mathtt{leaf} : P\ \mathtt{leaf}) \\ &\to (m_\mathtt{node} : (l\ r : \mathtt{Tree}) \to P\ l \to P\ r \to P\ (\mathtt{node}\ l\ r)) \\ &\to (x : \mathtt{Tree}) \to P\ x \end{aligned} \tag{87}$$

The evaluation rules are

$$\mathtt{elim}_\mathtt{Tree}\ P\ m_\mathtt{leaf}\ m_\mathtt{node}\ \mathtt{leaf} = m_\mathtt{leaf} \tag{88}$$

and

$$\begin{aligned} &\mathtt{elim}_\mathtt{Tree}\ P\ m_\mathtt{leaf}\ m_\mathtt{node}\ (\mathtt{node}\ l\ r) = \\ &\quad m_\mathtt{node}\ l\ r\ (\mathtt{elim}_\mathtt{Tree}\ P\ m_\mathtt{leaf}\ m_\mathtt{node}\ l)\ (\mathtt{elim}_\mathtt{Tree}\ P\ m_\mathtt{leaf}\ m_\mathtt{node}\ r) \end{aligned} \tag{89}$$

Case analysis $\mathtt{case}_\mathtt{D}$ is a weakened version of the standard eliminator without the inductive hypotheses.

Lemma 76. *We have a function* $\mathtt{case_D}$ *of type*

$$\begin{aligned}\mathtt{case_D}\ &: (P : \overline{\mathtt{D}} \to \mathtt{Set}_i)(m_1 : M_1)\dots(m_k : M_k)\\ &\to (\bar{x} : \overline{\mathtt{D}}) \to P\ \bar{x}\end{aligned} \tag{90}$$

where

$$\begin{aligned}M_i\ &: (\bar{t} : \Delta_i) \to (x_1 : \Phi_{i,1} \to D\ \bar{v}_{i,1})\ \dots\ (x_{n_i} : \Phi_{i,n_i} \to D\ \bar{v}_{i,n_i})\\ &\to P\ \bar{u}_i\ (\mathtt{c_i}\ \bar{t}\ x_1\ \dots\ x_{n_i})\end{aligned} \tag{91}$$

for $i = 1,\dots k$.

Example 77. *For the* `Tree` *type, we have*

$$\begin{aligned}&\mathtt{case_{Tree}}\ : (P : \mathtt{Tree} \to \mathtt{Set}_i) \to P\ \mathtt{leaf}\\ &\qquad\to ((l\ r : \mathtt{Tree}) \to P\ (\mathtt{node}\ l\ r)) \to (x : \mathtt{Tree}) \to P\ x\\ &\mathtt{case_{Tree}}\ P\ m_{\mathtt{leaf}}\ m_{\mathtt{node}}\ t = \mathtt{elim_{Tree}}\ P\ m_{\mathtt{leaf}}\ (\lambda l\ r\ h_l\ h_r.\ m_{\mathtt{node}}\ l\ r)\ t\end{aligned} \tag{92}$$

Example 78. *For the type* $m \leqslant n$, *we have*

$$\begin{aligned}\mathtt{case}_{\leqslant}\ &: (P : (m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to \mathtt{Set}_i)\\ &\to (m_{\mathtt{lz}} : (m : \mathbb{N}) \to P\ \mathtt{zero}\ m\ (\mathtt{lz}\ m))\\ &\to (m_{\mathtt{ls}} : (m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to P\ (\mathtt{suc}\ m)\ (\mathtt{suc}\ n)\ (\mathtt{ls}\ m\ n\ x))\\ &\to (m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to P\ m\ n\ x\end{aligned} \tag{93}$$

Construction of $\mathtt{case_D}$.

$$\mathtt{case_D}\ P\ m_1\ \dots\ m_k = \mathtt{elim_D}\ P\ (\lambda \bar{t}\ \bar{x}\ \bar{h}.\, m_1\ \bar{t}\ \bar{x})\ \dots\ (\lambda \bar{t}\ \bar{x}\ \bar{h}.\, m_k\ \bar{t}\ \bar{x}) \tag{94}$$

□

7.1 No confusion

Two of the unification rules, injectivity and conflict, are instances of a more general principle known as 'no confusion'. In this section, we construct this principle internally as an equivalence $\mathtt{noConf_D}$.

We first define an auxiliary type in order to give a general type to $\mathtt{noConf_D}$.

Lemma 79. *We have a type* $\mathtt{NoConfusion_D} : \overline{\mathtt{D}} \to \overline{\mathtt{D}} \to \mathtt{Set}_d$ *such that*

$$\begin{aligned}\mathtt{NoConfusion_D}\ (\bar{u}; \mathtt{c_i}\ \bar{s})\ (\bar{v}; \mathtt{c_i}\ \bar{t}) &= \bar{s} \equiv_{\Delta_i} \bar{t}\\ \mathtt{NoConfusion_D}\ (\bar{u}; \mathtt{c_i}\ \bar{s})\ (\bar{v}; \mathtt{c_j}\ \bar{t}) &= \bot \qquad (\textit{when } i \neq j)\end{aligned} \tag{95}$$

On the diagonal (where we have two times the same constructor), $\mathtt{NoConfusion_D}$ only requires $\bar{s} \equiv_{\Delta_c} \bar{t}$. From this, it follows that $\bar{u} \equiv_{\Xi} \bar{v}$ as well, since the indices are determined by the choice of constructor and its arguments.

Example 80. *For the* `Tree` *datatype,* `NoConfusion` $t_1\ t_2$ *is defined as follows:*

$$\begin{aligned}&\mathtt{NoConfusion} : \mathtt{Tree} \to \mathtt{Tree} \to \mathtt{Set}\\ &\mathtt{NoConfusion}\ \mathtt{leaf}\ \ \mathtt{leaf}\ \ = \top\\ &\mathtt{NoConfusion}\ \mathtt{leaf}\ \ (\mathtt{node}\ l\ r) = \bot\\ &\mathtt{NoConfusion}\ (\mathtt{node}\ l\ r)\ \ \mathtt{leaf}\ \ = \bot\\ &\mathtt{NoConfusion}\ (\mathtt{node}\ l_1\ r_1)\ (\mathtt{node}\ l_2\ r_2) = (l_1 \equiv_{\mathtt{Tree}} l_2) \times (r_1 \equiv_{\mathtt{Tree}} r_2)\end{aligned} \tag{96}$$

Construction of $\mathtt{NoConfusion_D}$. We apply $\mathtt{case_D}$ with the motive $\lambda\ \_.\ \overline{\mathtt{D}} \to \mathtt{Set}_i$. For each method $m_i\ \bar{x}$, we apply $\mathtt{case_D}$ again with motive $\lambda\_ \to \mathtt{Set}$. This gives us k^2 methods $m_{i,j}$ to fill in, one for each pair of constructors. On the diagonal (where $i = j$), we define $m_{ii} = \lambda\bar{x}; \bar{x}'.\ \bar{x} \equiv_{\Delta_i} \bar{x}'$, and if $i \neq j$, we give $m_{i,j} = \lambda\bar{x}; \bar{x}'.\ \bot$. □

Lemma 81. *We have an equivalence*

$$\mathtt{noConf_D} : (\bar{x}\ \bar{y} : \overline{\mathtt{D}}) \to (\bar{x} \equiv_{\overline{\mathtt{D}}} \bar{y}) \simeq \mathtt{NoConfusion_D}\ \bar{x}\ \bar{y} \tag{97}$$

Moreover, for any constructor $\mathtt{c} : \Delta \to \mathtt{D}\ \bar{u}$ *and* $\bar{s}, \bar{s}' : \Delta$, *this equivalence satisfies* $\mathtt{noConf_D}^{-1}\ (\bar{u}[\Delta \mapsto \bar{s}]; \mathtt{c}\ \bar{s})\ (\bar{u}[\Delta \mapsto \bar{s}']; \mathtt{c}\ \bar{s}') = \overline{\mathtt{dcong}}\ (\lambda\bar{x}.\bar{u}; \mathtt{c}\ \bar{x})$.

Example 82. *For* $\mathtt{Tree}$, *the function* $\mathtt{noConf_{Tree}}$ *gives for any two trees* s *and* t *that are equal a proof of* $\mathtt{NoConfusion_{Tree}}\ s\ t$:

$$\mathtt{noConf_{Tree}} : (s\ t : \mathtt{Tree}) \to (s \equiv_{\mathtt{Tree}} t) \simeq \mathtt{NoConfusion_{Tree}}\ s\ t \tag{98}$$

If s *and* t *are of the form* $\mathtt{node}\ l_1\ r_1$ *and* $\mathtt{node}\ l_2\ r_2$, *respectively, then this gives us the injectivity rule* $(\mathtt{node}\ l_1\ r_1 \equiv_{\mathtt{Tree}} \mathtt{node}\ l_2\ r_2) \simeq (l_1 \equiv_{\mathtt{Tree}} l_2 \times r_1 \equiv_{\mathtt{Tree}} r_2)$. *On the other hand, if* s *is of the form* $\mathtt{leaf}$ *and* t *is of the form* $\mathtt{node}\ l\ r$, *then we get the conflict rule* $(\mathtt{leaf} \equiv_{\mathtt{Tree}} \mathtt{node}\ l\ r) \simeq \bot$.

Construction of $\mathtt{noConf_D}$. First, we define the left-to-right function $\mathtt{noConf_D}\ \bar{a}\ \bar{b}$. To do this, we apply telescopic substitution $\overline{\mathtt{subst}}$ with motive $\mathtt{NoConfusion_D}\ \bar{a}$. This reduces the problem to finding a function of type

$$(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{NoConfusion_D}\ \bar{a}\ \bar{a} \tag{99}$$

But this can be done using $\mathtt{case_D}$ with motive $\lambda\ \bar{a}.\ \mathtt{NoConfusion_D}\ \bar{a}\ \bar{a}$, filling in $\overline{\mathtt{refl}}$ for each method $m_i\ \bar{x}$.

For the inverse $\mathtt{noConf_D}^{-1}\ \bar{a}\ \bar{b}$, we need to do a little more work. First, we apply $\mathtt{case_D}$ twice as in the definition of $\mathtt{NoConfusion_D}$. Now we are left to give methods

$$m_{i,j} : \mathtt{NoConfusion_D}\ (\bar{u}_i; \mathtt{c_i}\ \bar{x})\ (\bar{u}'_j; \mathtt{c_j}\ \bar{x}') \to \bar{u}_i\ (\mathtt{c_i}\ \bar{x}) \equiv_{\overline{\mathtt{D}}} \bar{u}'_j\ (\mathtt{c_j}\ \bar{x}') \tag{100}$$

When $i \neq j$, this is easy: we get an element of type $\bot$ from $\mathtt{NoConfusion_D}$, from which we can conclude anything. On the diagonal (where $i = j$), we get a proof of $\bar{x} \equiv_{\Delta_i} \bar{x}'$. Applying $\overline{\mathtt{dcong}}$ to this equality gives us $\bar{u}_i; (\mathtt{c_i}\ \bar{x}) \equiv_{\overline{\mathtt{D}}} \bar{u}'_i; (\mathtt{c_i}\ \bar{x}')$, which is what we need.

Next, we prove that this is a left inverse by constructing a function of type

$$(\bar{a}\ \bar{b} : \overline{\mathtt{D}})(\bar{e} : \bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b}) \to \mathtt{noConf_D}^{-1}\ \bar{a}\ \bar{b}\ (\mathtt{noConf_D}\ \bar{a}\ \bar{b}\ \bar{e}) \equiv_{\bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b}} \bar{e} \tag{101}$$

By $\overline{\mathtt{J}}$, it is sufficient to give a function of type

$$(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{noConf_D}^{-1}\ \bar{a}\ \bar{a}\ (\mathtt{noConf_D}\ \bar{a}\ \bar{a}\ \overline{\mathtt{refl}}) \equiv_{\bar{a} \equiv_{\overline{\mathtt{D}}} \bar{a}} \overline{\mathtt{refl}} \tag{102}$$

But this we can do by applying $\mathtt{case_D}$ with methods $m_i\ \bar{x} = \overline{\mathtt{refl}}$.

All that is left to do is to prove that it is a right inverse as well. To construct the proof $\mathtt{isRinv}$ that

$$(\bar{a}\ \bar{b} : \overline{\mathtt{D}})(e : \mathtt{NoConfusion_D}\ \bar{a}\ \bar{b}) \to \mathtt{noConf}\ \bar{a}\ \bar{b}\ (\mathtt{noConf}^{-1}\ \bar{a}\ \bar{b}\ e) \equiv_{\mathtt{NoConfusion_D}\ \bar{a}\ \bar{b}} e \tag{103}$$

we first apply case analysis on $\bar{a}$ and $\bar{b}$. In the cases where we have two distinct constructors $\mathtt{c_i}$ and $\mathtt{c_k}$, we have $e : \bot$ so we can conclude by $\mathtt{elim}_\bot$. In the diagonal cases, we have $e : \bar{s} \equiv_{\Delta_i} \bar{t}$. Eliminating these equations with $\mathtt{J}$ leaves us with the goal $\overline{\mathtt{refl}} \equiv_{\bar{s} \equiv_{\Delta_i} \bar{s}} \overline{\mathtt{refl}}$, which we solve by giving $\overline{\mathtt{refl}}$. □

7.2 *Acyclicity*

The other property of datatypes we need is acyclicity: a term can never be structurally smaller than itself. This property is used for implementing the cycle detection rule of the unification algorithm. Internally, it is represented by the term $\mathtt{noCycle_D}$. To express its type, we first define what it means for a term to (not) be structurally smaller than some other term.

We first define the auxiliary type $\mathtt{Below_D} : \mathtt{Below_D}\ P\ \bar{u}\ x$ is defined as a tuple type that is inhabited whenever $P\ \bar{v}\ y$ holds for all $y : \mathtt{D}\ \bar{v}$ that are structurally smaller than $x : \mathtt{D}\ \bar{u}$.

Lemma 83. *Let $P : \overline{\mathtt{D}} \to \mathtt{Set}_i$. For any $x : \mathtt{D}\ \bar{u}$, we have a type $\mathtt{Below_D}\ P\ \bar{u}\ x$ such that for any $y \prec x$ we have a projection $\pi : \mathtt{Below_D}\ P\ \bar{u}\ x \to P\ \bar{v}\ y$.*

Example 84. *The type $\mathtt{Below_{Tree}}\ P\ x$ expresses that the property $P : \mathtt{Tree} \to \mathtt{Set}$ holds for any subtree of $x : \mathtt{Tree}$. In other words, we have*

$$\begin{aligned} \mathtt{Below_{Tree}}\ P\ \mathtt{leaf} &= \top \\ \mathtt{Below_{Tree}}\ P\ (\mathtt{node}\ l\ r) &= (\mathtt{Below_{Tree}}\ P\ l \times P\ l) \times (\mathtt{Below_{Tree}}\ P\ r \times P\ r) \end{aligned} \tag{104}$$

Construction of $\mathtt{Below_D}\ P$. We apply the eliminator $\mathtt{elim_D}$ to the motive $\Phi = \lambda\ \_.\ \mathtt{Set}_i$. For the method m_i corresponding to the constructor $\mathtt{c_i}$, we give the following:

$$\begin{aligned} m_i = \lambda \bar{t}; x_1; \ldots; x_{n_i}; h_1; \ldots; h_{n_i}. \\ ((\bar{s}_1 : \Phi_{i,1}) \to h_1\ \bar{s}_1 \times P\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \times \\ \cdots \times ((\bar{s}_{n_i} : \Phi_{i,n_i}) \to h_{n_i}\ \bar{s}_{n_i} \times P\ \bar{v}_{i,n_i}\ (x_{n_i}\ \bar{s}_{n_i})) \end{aligned} \tag{105}$$

To construct the projection π, consider $x : \mathtt{D}\ \bar{u}$ and any structurally smaller term $y : \mathtt{D}\ \bar{v}$. If y is (an application of) a direct subterm of x, say $x = \mathtt{c}\ \bar{t}\ x_1\ \ldots\ x_n$ with $y = x_i\ \bar{w}$, then we return the second component of the ith component of $\mathtt{Below_D}\ P\ x$, i.e. we define

$$\pi\ H = \pi_2\ (\pi_i\ H\ \bar{w}) : \mathtt{Below_D}\ P\ \bar{u}\ x \to P\ \bar{v}\ y \tag{106}$$

Otherwise, y is a subterm of some direct subterm x_i of $x = \mathtt{c}\ \bar{t}\ x_1\ \ldots\ x_n$. In particular, by induction, we have some $\pi' : \mathtt{Below_D}\ P\ \bar{v}_i\ x_i \to P\ \bar{v}\ y$. This allows us to define π as follows:

$$\pi\ H = \pi'\ (\pi_1\ (\pi_i\ H)) : \mathtt{Below_D}\ P\ \bar{u}\ x \to P\ \bar{v}\ y \tag{107}$$

□

Lemma 85. *We have a type $\_ \not\prec_\mathtt{D} \_ : \overline{\mathtt{D}} \to \overline{\mathtt{D}} \to \mathtt{Set}_d$ such that for any $x : \mathtt{D}\ \bar{u}$ and $y : \mathtt{D}\ \bar{v}$ with $x \prec y$, we have $x \not\prec_\mathtt{D} y \to \bot$. We also define $\bar{a} \not\preccurlyeq_\mathtt{D} \bar{b} := \bar{a} \not\prec_\mathtt{D} \bar{b} \times \bar{a} \not\equiv_\mathtt{D} \bar{b}$.*

If $x : \mathtt{D}\ \bar{u}$ and $y : \mathtt{D}\ \bar{v}$, then we often leave the indices implicit and write $x \not<_{\mathtt{D}} y$ and $x \not\leqslant_{\mathtt{D}} y$ instead of $\bar{u}; x \not<_{\mathtt{D}} \bar{v}; y$ and $\bar{u}; x \not\leqslant_{\mathtt{D}} \bar{v}; y$.

Example 86. *The type $x \not<_{\mathtt{Tree}} t$ expresses that x is* not *a subtree of t. In particular, we have the following equalities:*

$$\begin{aligned} x \not<_{\mathtt{Tree}} \mathtt{leaf} &= \top \\ x \not<_{\mathtt{Tree}} (\mathtt{node}\ l\ r) &= ((x \not<_{\mathtt{Tree}} l) \times (x \not\equiv_{\mathtt{Tree}} l)) \times ((x \not<_{\mathtt{Tree}} r) \times (x \not\equiv_{\mathtt{Tree}} r)) \end{aligned} \tag{108}$$

Construction of $\not<_{\mathtt{D}}$. We define $\not<_{\mathtt{D}}$ in terms of $\mathtt{Below}_{\mathtt{D}}$:

$$\bar{a} \not<_{\mathtt{D}} \bar{b} := \mathtt{Below}_{\mathtt{D}}\ (\lambda \bar{b}'.\ \bar{a} \not\equiv \bar{b}')\ \bar{b} \tag{109}$$

By definition of $\mathtt{Below}_{\mathtt{D}}$, we have a projection $\pi : \bar{u}; x \not<_{\mathtt{D}} \bar{v}; y \to \bar{u}; x \not\equiv_{\overline{\mathtt{D}}} \bar{u}; x$, whenever $x \prec y$. Filling in $\overline{\mathtt{refl}}$ for the proof of $\bar{u}; x \not\equiv_{\overline{\mathtt{D}}} \bar{u}; x$ gives us the desired proof of $x \not<_{\mathtt{D}} y \to \bot$. □

Now we can state the property that no term can be structurally smaller than itself.

Lemma 87. *We have a function $\mathtt{noCycle}_{\mathtt{D}} : (\bar{a}\ \bar{b} : \overline{\mathtt{D}}) \to \bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b} \to \bar{a} \not<_{\mathtt{D}} \bar{b}$.*

Example 88. *$\mathtt{noCycle}_{\mathtt{Tree}}$ is the proof that no tree can ever be a subtree of itself, i.e. every well-typed tree is well-founded.*

Construction of $\mathtt{noCycle}_{\mathtt{D}}$. Note that

$$\begin{aligned} x \not<_{\mathtt{D}} \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} = ((\bar{s}_1 : \Phi_{i,1}) \to x \not\leqslant_{\mathtt{D}} x_1\ \bar{s}_1) \times \ldots \\ \times ((\bar{s}_{n_i} : \Phi_{i,n_i}) \to x \not\leqslant_{\mathtt{D}} x_{n_i}\ \bar{s}_{n_i}) \end{aligned} \tag{110}$$

by definition of $\mathtt{Below}_{\mathtt{D}}$ and $\not\leqslant_{\mathtt{D}}$. Now to construct $\mathtt{noCycle}_{\mathtt{D}}$, we start by eliminating the equation $\bar{a} \equiv_{\overline{\mathtt{D}}} \bar{b}$ using $\overline{\mathtt{J}}$, which leaves us the goal $(\bar{a} : \overline{\mathtt{D}}) \to \bar{a} \not<_{\mathtt{D}} \bar{a}$. Next, we apply $\mathtt{case}_{\mathtt{D}}$ with motive $\lambda \bar{a}.\ \bar{a} \not<_{\mathtt{D}} \bar{a}$, producing for each constructor $\mathtt{c_i} : \Delta_i \to (\Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1}) \to \ldots \to (\Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \mathtt{D}\ \bar{u}_i$ the subgoal

$$\begin{aligned} &(\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ &\quad (h_1 : (\bar{s}_1 : \Phi_{i,1}) \to x_1\ \bar{s}_1 \not<_{\mathtt{D}} x_1\ \bar{s}_1)\ \ldots \\ &\quad (h_{n_i} : (\bar{s}_{n_i} : \Phi_{i,n_i}) \to x_{n_i}\ \bar{s}_{n_i} \not<_{\mathtt{D}} x_{n_i}\ \bar{s}_{n_i}) \to \\ &\quad \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not<_{\mathtt{D}} \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \end{aligned} \tag{111}$$

To continue, we define the auxiliary types $\mathtt{Step_{i,j}}$ for $i = 1, \ldots, k$ and $j = 1, \ldots, n_i$ as follows:

$$\begin{aligned} &\mathtt{Step_{i,j}} : (\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ &\qquad (\bar{s} : \Phi_{i,j})(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{Set}_d \\ &\mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ (\bar{u}; b) = (x_j\ \bar{s}) \not<_{\mathtt{D}} b \to (\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \not\leqslant_{\mathtt{D}} b \end{aligned} \tag{112}$$

In what follows, we will construct

$$\begin{aligned} \mathtt{step_{i,j}} : &(\bar{t} : \Delta_i)(x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ &(\bar{s} : \Phi_{i,j})(\bar{a} : \overline{\mathtt{D}}) \to \mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \Phi_{i,j}\ \bar{a} \end{aligned} \tag{113}$$

Once this is done, we solve the subgoal (111) by filling in

$$\begin{aligned} &\lambda \bar{t}; x_1; \ldots; x_{n_i}; h_1; \ldots; h_{n_i}. \\ &\quad (\lambda \bar{s}_1.\ \mathtt{step_{i,1}}\ \bar{t}\ \bar{x}\ \bar{s}_1\ (\bar{v}_{i,1}; (x_1\ \bar{s}_1))\ (h_1\ \bar{s}_1)), \ldots, \\ &\quad (\lambda \bar{s}_{n_i}.\ \mathtt{step_{i,n_i}}\ \bar{t}\ \bar{x}\ \bar{s}_{n_i}\ (\bar{v}_{i,n_i}; (x_{n_i}\ \bar{s}_{n_i}))\ (h_{n_i}\ \bar{s}_{n_i})) \end{aligned} \tag{114}$$

So we only need to construct $\mathtt{step}_{\mathtt{i,j}}$.

The construction of $\mathtt{step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s} : (\bar{a} : \bar{\mathtt{D}}) \to \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{a}$ proceeds by applying $\mathtt{elim}_{\mathtt{D}}$ with the motive $\mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}$. The new subgoals are of the form

$$\begin{aligned}&(\bar{t}' : \Delta_p)(x'_1 : \Phi_{p,1} \to \mathtt{D}\ \bar{v}'_{p,1})\ \ldots\ (x'_{n_p} : \Phi_{p,n_p} \to \mathtt{D}\ \bar{v}'_{p,n_p}) \to \\ &\quad (h'_1 : (\bar{s}'_1 : \Phi_{p,1}) \to \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{v}'_{p,1}\ (x'_1\ \bar{s}'_1))\ \ldots \\ &\quad (h'_{n_p} : (\bar{s}'_{n_p} : \Phi_{p,n_p}) \to \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{v}'_{p,n_p}\ (x'_{n_p}\ \bar{s}'_{n_p})) \to \\ &\quad \mathtt{Step}_{\mathtt{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \bar{s}\ \bar{u}'_p\ (\mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p})\end{aligned} \tag{115}$$

We solve them by giving

$$\lambda \bar{t}'; x'_1; \ldots; x'_{n_p}; h'_1; \ldots; h'_{n_p}; H.\ \alpha, \beta \tag{116}$$

where we still have to construct

$$\alpha : \bar{u}_i; (\mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \not<_{\mathtt{D}} \bar{u}'_p; (\mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p}) \tag{117}$$

and

$$\beta : \bar{u}_i; (\mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \not\leq_{\bar{\mathtt{D}}} \bar{u}'_p; (\mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p}) \tag{118}$$

We have $H : x_j\ \bar{s} \not<_{\mathtt{D}} \mathtt{c}_{\mathtt{p}}\ \Delta'_p\ x'_1\ \ldots\ x'_{n_p}$ or, by definition of $\not<_{\mathtt{D}}$, $H = (H_1, \ldots, H_{n_p})$, where $H_q : (\bar{s}' : \Phi'_{pq}) \to x_j\ \bar{s} \not\leq_{\mathtt{D}} x'_q\ \bar{s}'$. The construction of α reduces to the construction of components $\alpha_q : (\bar{s}' : \Phi'_{p,q}) \to \mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not<_{\mathtt{D}} x'_q\ \bar{s}'$. But these we can give as $\alpha_q = \lambda \bar{s}'.\ h'_q\ \bar{s}'\ (\pi_1\ (H_p\ \bar{s}'))$.

For constructing β, we assume $\bar{u}_i; (\mathtt{c}_{\mathtt{i}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \equiv_{\bar{\mathtt{D}}} \bar{u}'_p; (\mathtt{c}_{\mathtt{p}}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p})$ and derive an element of $\bot$. By $\mathtt{noConf}_{\mathtt{D}}$, it suffices to consider the case where $i = p$ and $\bar{t}; x_1; \ldots; x_{n_i} = \bar{t}'; x'_1, \ldots, x'_{n_p}$. But then we have $H_j\ \bar{s} : x_j\ \bar{s} \not\leq_{\mathtt{D}} x_j\ \bar{s}$, hence $\pi_2\ (H_j\ \bar{s})\ \mathtt{refl} : \bot$. This finishes the construction of $\mathtt{noCycle}_{\mathtt{D}}$. □

8 Implementation

Using our framework for proof-relevant unification described in this paper, we reimplemented the unification algorithm used by Agda for checking definitions by dependent pattern matching. As a result, we were able to replace previous *ad-hoc* restrictions with formally verified unification rules, fixing a number of bugs in the process. It also enabled us to add new unification rules dealing with η-equality for record types, as well as higher dimensional unification for solving equations between constructors of indexed datatypes. Another advantage of our approach is that the implementation is now much cleaner than before, allowing it to be extended easily in the future. In this section, we take a look at our implementation from the point of view of an Agda user (Section 8.1) and an Agda developer (Section 8.2).

8.1 Impact on the Agda user

From the point of view of a user of Agda, unification happens behind the scenes while checking definitions by pattern matching, so a different algorithm does not impact the syntax of the language directly. Instead, the main criterion a user of Agda should judge the unification algorithm by is that it accepts the definitions that should

be accepted, and rejects the definitions that should be rejected. The latter can be seen from the fact that our implementation directly resulted in a fix for issue #1408 (Example 41), dealing with an incompatibility between heterogeneous equations and the—without-K option (Vezzosi, 2015). Equally important, our implementation provides a much more principled solution to issues #292 (Danielsson, 2010, see also Example 41), #1071 (Danielsson, 2014), #1406 (Abel, 2015a, see also Example 40), #1411 (Abel, 2015b), and #1427 (Abel, 2015c, see also Example 40). All these issues are fixed without introducing special cases in the code and without limiting the power of the unification algorithm in any significant way, as can be seen from the fact that Agda's test suite and standard library are still typechecked correctly. This is in contrast to the previous *ad-hoc* fixes to some of these issues, which broke the unification algorithm in some cases, for example, in issue #1435 (Danielsson, 2015).

The addition of the new unification rules for η-equality of record values also significantly improved the way Agda handles records. Before these unification rule were added to Agda, all variables of record type had to be fully eta-expanded before calling the unifier, for example, in issue #473 (Danielsson, 2011). This caused a substantial overhead when dealing with deeply nested records, see issue #635 (Peebles, 2012). This also caused problems in combination with Agda's instance search mechanism, see, for example, issue #1613 (Abel, 2015d). In contrast, by using this unification rule, we only eta-expand a variable when it is useful for the unification to proceed, thus eliminating this overhead.

We also implemented higher dimensional unification (Section 6). This addition allows Agda to typecheck more definitions, such as the example given in issue #1775 (Sicard-Ramírez, 2016).

8.2 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 12. The unification strategy takes a unification state as an argument and produces a lazy monadic list of unification rules to try (Figure 13), while the unification engine tries to apply these rules one by one until one succeeds (Figure 14).

A big difference between our implementation and Agda's previous unification algorithm is that our version explicitly manipulates telescopes of free variables (`varTel`) and equations (`eqTel`) as well as explicit substitutions between these telescopes, while previously these had to be reconstructed after unification was finished. This change resulted in a significant simplification of the code for checking left-hand sides and coverage of definitions by pattern matching (the parts of Agda that use the unification algorithm).

An important choice when constructing a unification strategy is whether to start on the leftmost or the rightmost equation. It seems sensible to start on the left to avoid heterogeneous equations as much as possible, and this was also the preferred

```
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      { ... }
```

Fig. 12. The datatypes 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, 2012), 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.

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

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

Fig. 13. 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.

method for the old algorithm. However, our unification rules for indexed datatypes actually benefit from having unsolved equations in the telescope, so a unification strategy that starts from the right provides more opportunities to apply these rules. For this reason, our current implementation uses a right-to-left strategy, although plugging in a different strategy would be trivial.

Our implementation of higher dimensional unification closely follows the steps in Section 6.2. In particular, when applying the injectivity rule to a unification problem of the form $(\bar{e}_1 : \bar{s}_1 \equiv_\Phi \bar{s}_2)(e_2 : \mathtt{c}\ \bar{t}_1 \equiv_{\mathtt{D}\ \bar{v}_e} \bar{t}_2)$, the unification algorithm constructs the new unification problem $\Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$ and recursively calls itself on this new problem.

One noteworthy fact about the implementation is how the left- and right-hand sides $f\ \bar{s}_1\ \bar{t}_1\ \overline{\mathtt{refl}}$ and $f\ \bar{s}_2\ \bar{t}_2\ \overline{\mathtt{refl}}$ of the new unification problem in Equation (73)

```
unifyStep :: UnifyState -> UnifyStep
          -> UnifyM (UnificationResult' UnifyState)

unify :: UnifyState -> UnifyStrategy
      -> UnifyM (UnificationResult' UnifyState)
```

Fig. 14. 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.

are computed. The implementation does not have an explicit representation of the function f, so it is not possible to calculate them directly. Instead, the recursive call produces a substitution ρ of type $\Delta' \to \Phi\Delta$. This allows us to calculate $f^{-1} : \Delta' \to (\bar{x} : \Phi)(\bar{y} : \Delta)(\bar{p} : \bar{u} \equiv_\Xi \bar{v})$ as $\lambda\bar{x}'.\ \bar{x}'\rho;\overline{\texttt{refl}}$, but does not give us a direct way to compute f. To go in the opposite direction, we note that ρ is a *pattern* with free variables Δ'. So we can match the values from $\Phi\Delta$ against this pattern. The proofs of $\bar{u} \equiv_\Xi \bar{v}$ (assumed to be $\overline{\texttt{refl}}$ in our implementation, since all the unifiers we compute are strong unifiers) ensure that this matching cannot fail, so this allows us to recover the values of the variables in Δ', thus computing the function $f : \Phi\Delta(\bar{p} : \bar{u} \equiv_\Xi \bar{v}) \to \Delta'$.

9 Related work

Unification is a large area of research that we cannot hope to cover here in full. We refer the interested reader to Jouannaud and Kirchner (1991) and Baader and Snyder (2001) for a general overview of the subject. Most extensions to unification that are studied, such as higher order unification and E-unification, are orthogonal to the work in this paper, although it would be interesting to see how they fit within our framework.

Type checkers of dependently typed languages typically have some facility for metavariables that are solved by higher order pattern unification (Reed, 2009; Abel and Pientka, 2011). This is not directly related to the work in this paper as the requirements on the unification algorithm are 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 (Ziliani and Sozeau, 2015) or make use of user-provided hints to choose one solution over the other (Asperti, Ricciotti, Sacerdoti Coen, and Tassi, 2009), thereby giving up on finding most general unifiers in favour of finding solutions more often. In this case, the only problem is that the solution to the metavariable may not be what the user intended. In contrast, our algorithm produces evidence of unification internal to the theory we are working in, and it is actually important that the unifier found by the algorithm is indeed the most general one (otherwise we might lose e.g. coverage of functions by pattern matching). Still, it would be interesting to further investigate the similarities and differences between these two unification algorithms.

Goguen (1989) takes a categorical view on unification, representing most general unifiers as *equalizers* in a category of types and substitutions. It should not be surprising that many of the category-theoretic notions are 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 datatypes, and it also differs in the fact that type theory allows an internal representation of equations as (telescopic) equality types.

The idea to represent unification problems at the object level by using the identity type stems from McBride (1998b). In McBride's paper, the types of equations are limited to simple (non-dependent) types, and the injectivity rule is likewise limited to simple datatypes. Later, he solves this by introducing a heterogeneous identity type (McBride, 2002). However, UIP is needed to turn heterogeneous equalities back into homogeneous ones. Additionally, postponing equations is not supported, as heterogeneous equations can only be turned into homogeneous ones if the types are equal. In our previous work, we solved the problem of requiring UIP, but the unification rules still only worked on the first equation in a telescope (Cockx *et al.*, 2014). As a consequence, we had to limit the injectivity, conflict, and cycle rules to work only in homogeneous situations, while here we can use them in their fully general form.

Our approach to unification is closely related to the notion of inversion of an inductive hypothesis (Cornes and Terrasse, 1996; Monin, 2010). The usual approach to inversion works by crafting a *diagonalizer* that is used as the motive for an eliminator. Unification can also be as an alternative method for proving inversion lemmas (McBride, 1998b). One advantage of the diagonalizer approach is that it moves most of the work to the type level, potentially improving performance of the resulting function. The process of constructing diagonalizers has recently also been automated (Braibant, 2013). However, it requires that the indices of the inductive hypothesis we are inverting can be written as a pattern, which is not always the case (e.g. they may be non-linear), so the approach based on unification seems to be more general. It would be interesting to try to implement an inversion tactic based on the unification algorithm in this paper to compare the power of the two approaches.

The idea to view equality proofs themselves as the subjects of unification is inspired by cubical type theory, where equality proofs are terms viewed 'one level up' (Cohen *et al.*, 2016). In fact, if we were working in a cubical type theory, there would be no difference between regular unification and higher dimensional unification, so the work in this paper could be seen as 'backporting' some of the power of cubical type theory back to the (currently) better-understood world of standard intuitionistic type theory.

Compared to our reverse unification rules from Cockx *et al.* (2016a), higher dimensional unification takes information into account from the types of the constructors as well as the types of the equation. This difference is similar to the inversion of an inductive hypothesis by using a diagonalizer (Cornes and Terrasse, 1996) versus using unification for the problem (McBride, 1998b).

10 Discussion and future work

In this paper, we present a proof-relevant unification algorithm for use in dependent type theory, where unification rules are represented as terms internal to the type theory. Thus, the type system itself enforces the soundness of these unification rules. Moreover, this lets us extend the unification algorithm with new principles in a safe and modular way. For example, we showed how to add two new unification rules for η-equality of record types. As another example, higher dimensional unification augments the power of the injectivity rule by allowing us to skip unification of forced arguments, yet would be impossible to even formulate for an untyped unification algorithm.

Having an elegant theoretical framework for unification also helped us a lot when implementing it in practice. As a result, the implementation of our algorithm for Agda has become cleaner, more robust, and more easily extensible. We hope this will also be the case for implementers of other dependently typed languages, as it has already been for the Lean theorem prover and the Equations package for Coq.

Other applications of proof-relevant unification. In this work, we focus on one application of proof-relevant unification, namely specialization by unification and its role in the compilation of dependent pattern matching. However, we believe firmly that it could also be applied elsewhere, for example, for metaprogramming or tactic systems.

More unification rules. It would be interesting to further explore the correspondence between unification rules and new features of type theory. For example, it seems that E-unification (unification modulo a set of equations) could correspond to new unification rules for higher inductive types from HoTT. As another example, higher order (pattern) unification could correspond to functional extensionality as a unification rule. And since the univalence axiom is itself an equivalence, maybe it could be seen as a unification rule as well?

Custom unification rules. We can put the power of unification in the hands of the user by allowing them to define custom unification rules in the form of *hints* (Asperti et al., 2009). For example, if the user provides a proof of $(f\ x \equiv_B f\ y) \simeq (x \equiv_A y)$ for some function $f : A \to B$, then this could be used as an injectivity rule for f by the unifier. One obstacle is that these rules might not be strong unification rules, so we either have to give up on some computational properties of unifiers or implement a check of strongness of a given unification rule.

Unification with higher inductive types. HoTT introduces the concept of *higher inductive types*, which can have non-trivial identity proofs between their constructors. This implies that in general they do not satisfy the injectivity, disjointness, or acyclicity properties. So to adapt our unification algorithm to a context with higher inductive types, we should start by limiting the unification algorithm further, for example, by cutting out the 'no confusion' and 'cycle' properties for types to which they do not apply.

As a second step, these principles can be replaced by type-specific solvers that exploit any extra structure that may be available.

Example 89. *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:*

$$\mathtt{contract} : (e : \mathtt{0} \equiv_{\mathtt{I}} \mathtt{1}) \simeq () \tag{119}$$

By definition, we have $\mathtt{contract}^{-1}\ () = \mathtt{line}$*, so if we use this equivalence as a unification rule, we will not 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 do not have the proper definitional behaviour, but still produce an equivalence of the correct type.*

More generally, the 'no confusion' principle is similar to the encode/decode technique used by Licata and Shulman (2013) and McKinna and Forsberg (2015) to calculate the fundamental group of the circle. In particular, they also construct an equivalence between an equality/path type and a type of *codes* taking the role of our `NoConfusion` type. So it may be possible to construct a new unification rule for the circle type based on this equivalence. However, be aware that these custom unification rules can introduce additional variables, for example, the rule for the circle introduces a variable of type $\mathbb{Z}$!

Unification in cubical type theory. Our unification algorithm is developed for an Agda-like theory based on standard MLTT. In such a theory, principles such as functional extensionality or univalence can be postulated but they do not get any computational behaviour. On the other hand, a new and promising theory called *cubical* type theory gives a constructive interpretation to the univalence axiom, and hence also functional extensionality (Bezem, Coquand, and Huber, 2014; Cohen *et al.*, 2016). In the future, we would like to adapt the work in this paper to this setting, so it would become usable in languages based on cubical type theory as well.

One obstacle for this adaptation is the fact that the representation of datatypes in our theory (and also that of Agda, Coq, Idris, . . .) is computationally incompatible with functional extensionality. We give an example to illustrate the problem.[7]

Example 90. *Let* $\mathtt{Favourite} : (\mathbb{N} \to \mathbb{N}) \to \mathtt{Set}$ *be a datatype with one constructor* $\mathtt{favourite} : \mathtt{Favourite}\ (\lambda x.\ 0+x)$*. We can give a proof* p *of* $(x : \mathbb{N}) \to 0+x \equiv x+0$*, so we have* $\mathtt{funext}\ p : \lambda x.\ 0 + x \equiv \lambda x.\ x + 0$ *and thence*

$$\mathtt{subst}\ \mathtt{Favourite}\ (\mathtt{funext}\ p)\ \mathtt{favourite} : \mathtt{Favourite}\ (\lambda x.\ x + 0) \tag{120}$$

However, there is no closed canonical form of type $\mathtt{Favourite}\ (\lambda x.\ x+0)$*, so this term does not reduce to a canonical form. This cannot be fixed by taking the constructor itself to be the canonical form (i.e. by letting* $\mathtt{favourite} : \mathtt{Favourite}\ (\lambda x.\ x + 0)$*),*

[7] Thanks to Conor McBride for pointing out the problem and giving this example.

as this would require the typechecker to check whether two functions are extensionally equal, which is undecidable in general.

This incompatibility could be solved by disallowing indexed datatypes and instead having each constructor carry explicit proofs of the constraints it imposes on the former indices. For example, `favourite` would have the internal type $(e : f \equiv (\lambda x.\ 0 + x)) \to \texttt{Favourite}\ f$. The surface-level constructor is then represented as `favourite refl`, while `subst Favourite (funext p) favourite` computes to `favourite (funext p)`. With this representation of datatypes, the work done in this paper is just as necessary as before, since we still need unification to solve the (telescopic) equations embedded in the constructors, as well as equations between these embedded equality proofs.

Example 91. *We illustrate this by working out Example 63 again for a version of the* `Vec` *datatype with embedded equality proofs instead of indices. Suppose* $\texttt{Vec}\ A\ n$ *is defined with constructors* $\texttt{nil} : n \equiv_{\mathbb{N}} \texttt{zero} \to \texttt{Vec}\ A\ n$ *and* $\texttt{cons} : (m : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ m) \to n \equiv_{\mathbb{N}} \texttt{suc}\ m \to \texttt{Vec}\ A\ n$ *and consider the unification problem:*

$$(e : \texttt{cons}\ n\ x\ xs\ \texttt{refl} \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)} \texttt{cons}\ n\ y\ ys\ \texttt{refl}) \tag{121}$$

Since this version of the `Vec` *datatype does not have an index, we can apply the* $\texttt{injectivity}_{\texttt{cons}}$ *rule to simplify this equation to*

$$\begin{aligned}&(e_1 : n \equiv_{\mathbb{N}} n)(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\texttt{Vec}\ A\ e_1} ys)\\ &(e_4 : \texttt{refl} \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ e_1} \texttt{refl})\end{aligned} \tag{122}$$

Now, e_4 *is an equation between equality proofs, much like the one we obtained in (69), except that the equality* $(p : \texttt{suc}\ e_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{suc}\ n)$ *is replaced with an equality* $(e_4 : \texttt{refl} \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ e_1} \texttt{refl})$*. Lemma 71 shows that these two types are in fact equivalent. So higher dimensional unification problems also occur in languages without indexed datatypes, and hence that a general way to solve this kind of equations is equally useful in these languages.*

Acknowledgments

We want to thank the anonymous reviewers for their hard work and insightful comments. Dominique Devriese holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO).

References

Abel, A. (2011) Irrelevance in type theory with a heterogeneous equality judgement. In *Foundations of Software Science and Computational Structures,* Hofmann, M. (ed). Berlin Heidelberg: Springer, pp. 57–71. ISBN 978-3-642-19805-2.

Abel, A. (2012) MiniAgda: Integrating sized and dependent types. In *PAR-10. Partiality and Recursion in Interactive Theorem Provers*, volume 5 of *EPiC Series in Computing*, Ekaterina, K., Ana, B. & Milad, N. (eds). EasyChair, pp. 18–33. Available at: `https://easychair.org/publications/paper/RM`.

Abel, A. (2015a) Injectivity of type constructors is partially back. Agda refutes excluded middle. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1406` (on the Agda bug tracker).

Abel, A. (2015b) Order of patterns matters for checking left hand sides. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1411` (on the Agda bug tracker).

Abel, A. (2015c) Circumvention of forcing analysis brings back easy proof of Fin injectivity. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1427` (on the Agda bug tracker).

Abel, A. (2015d) Eta-expanded implicit patterns are not used for instance search. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1613` (on the Agda bug tracker).

Abel, A. & Pientka, B. (2011) Higher-order dynamic pattern unification for dependent types and records. In International Conference on Typed Lambda Calculi and Applications, TLCA. Springer, pp. 10–26.

Asperti, A., Ricciotti, W., Sacerdoti Coen, C., & Tassi, E. (2009) Hints in unification. In *Theorem Proving in Higher Order Logics*, Berghofer, S., Nipkow, T., Urban, C. & Wenzel, M. (eds). Berlin, Heidelberg: Springer, pp. 84–98. ISBN 978-3-642-03359-9.

Baader, F. & Snyder, W. (2001) Unification theory. In *Handbook of Automated Reasoning (in 2 volumes)*, Robinson, J. A. & Voronkov, A. (eds). Elsevier and MIT Press, pp. 445–532. ISBN 0-444-50813-9.

Bezem, M., Coquand, T., & Huber, S. (2014) A Model of Type Theory in Cubical Sets. In 19th International Conference on Types for Proofs and Programs (TYPES 2013), volume 26 of Leibniz International Proceedings in Informatics (LIPIcs), Matthes, R. & Schubert, A. (eds). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 107–128. ISBN 978-3-939897-72-9. Accessed April 17, 2017. Available at: `http://drops.dagstuhl.de/opus/volltexte/2014/4628`.

Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* **23**(5): 552–593.

Brady, E., McBride, C., & McKinna, J. (2004) Inductive families need not store their indices. In *Types for Proofs and Programs*, Berardi, S., Coppo, M., & Damiani, F. (eds). Berlin, Heidelberg: Springer, pp. 115–29. ISBN 978-3-540-24849-1.

Braibant, T. (2013) A new Coq tactic for inversion. Accessed April 17, 2017. Available at: `http://gallium.inria.fr/blog/a-new-Coq-tactic-for-inversion`

Cockx, J. (2017) *Dependent Pattern Matching and Proof-Relevant Unification*. PhD Thesis, KU Leuven.

Cockx, J. & Devriese, D. (2017) Lifting proof-relevant unification to higher dimensions. In Proceedings of the 6th Conference on Certified Programs and Proofs, CPP. ACM.

Cockx, J., Devriese, D. & Piessens, F. (2014) Pattern matching without K. In Proceedings of the 19th International Conference on Functional Programming, ICFP. ACM.

Cockx, J., Devriese, D. & Piessens, F. (2016a) Unifiers as equivalences: Proof-relevant unification of dependently typed data. In Proceedings of the 21th International Conference on Functional Programming, ICFP. ACM.

Cockx, J., Devriese, D., & Piessens, F. (2016b) Eliminating dependent pattern matching without K. *J. Funct. Program.* **26**, e16.

Cohen, C., Coquand, T., Huber, S. & Mörtberg, A. (2016) Cubical type theory: A constructive interpretation of the univalence axiom. In *CoRR*. Accessed April 17, 2017. Available at: `http://arxiv.org/abs/1611.02108`

Coquand, T. (1992) Pattern matching with dependent types. In Proceedings of the 1992 Workshop on Types for Proofs and Programs, TYPES, Nordstrom, B., Petersson, K., & Plotkin, G. (eds). Chalmers University of Technology, pp. 66–79.

Cornes, C. & Terrasse, D. (1996) Automating inversion of inductive predicates in Coq. In *Types for Proofs and Programs*, Berardi, S. & Coppo, M. (eds). Berlin, Heidelberg: Springer, pp. 85–104. ISBN 978-3-540-70722-6.

Danielsson, N. A. (2010) Heterogenous equality is crippled by the Bool /= Fin 2 fix. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/292` (on the Agda bug tracker).

Danielsson, N. A. (2011) The unification machinery does not respect η-equality. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/473` (on the Agda bug tracker).

Danielsson, N. A. (2014) Regression in unifier, possibly related to modules and/or heterogeneous constraints. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1071` (on the Agda bug tracker).

Danielsson, N. A. (2015) Dependent pattern matching is broken. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1435` (on the Agda bug tracker).

de Bruijn, N. G. (1991) Telescopic mappings in typed lambda calculus. *Inf. Comput.* **91**(2), 189–204. ISSN 0890-5401. Accessed April 17, 2017. Available at: `http://www.sciencedirect.com/science/article/pii/089054019190066B`

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

Dijkstra, G. (2015) Disunifying non-fully applied constructors is inconsistent with function extensionality. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1497` (on the Agda bug tracker).

Dybjer, P. (1991) Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In Proceedings of the 1st Workshop on Logical Frameworks.

Goguen, H., McBride, C., & McKinna, J. (2006) Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Futatsugi, K., Jouannaud, J.-P., & Meseguer, J. (eds). Berlin, Heidelberg: Springer, pp. 521–540. ISBN 978-3-540-35464-2.

Goguen, Joseph A. (1989) What is unification?: A categorical view of substitution, equation and solution. In *Algebraic Techniques*, Aït-Kaci, H. & Nivat, M. (eds). Academic Press, pp. 217–261. ISBN 978-0-12-046370-1.

Hur, C. K. (2010) Agda with the excluded middle is inconsistent?. Accessed April 17, 2017. Available at: `https://lists.chalmers.se/pipermail/agda/2010/001522.html` (On the Agda mailing list).

Jouannaud, J.-P. & Kirchner, C. (1991) Solving equations in abstract algebras: A rule-based survey of unification. In *Computational Logic - Essays in Honor of Alan Robinson*, Lassez, J.-L. & Plotkin, G. D. (eds). The MIT Press, pp. 257–321.

Licata, D. R. & Shulman, M. (2013) Calculating the fundamental group of the circle in homotopy type theory. In 28th Symposium on Logic in Computer Science, LICS.

Luo, Z. (1994) *Computation and Reasoning: A Type Theory for Computer Science*, volume 11 of *International Series of Monographs on Computer Science*. New York, NY, USA: Oxford University Press, Inc. ISBN 0-19-853835-9.

Martin-Löf, P. (1972) An intuitionistic theory of types. In Proceedings of the 25th Years of Constructive Type Theory (Venice, 1995). Oxford University Press, pp. 127–172.

Martin-Löf, P. (1984) *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis. ISBN 88-7088-105-9.

McBride, C. (1998a) Towards dependent pattern matching in LEGO. Unpublished.

McBride, C. (1998b) Inverting inductively defined relations in LEGO. In *Types for Proofs and Programs*, Giménez, E. & Paulin-Mohring, C. (eds). Berlin, Heidelberg: Springer, pp. 236–253. ISBN 978-3-540-49562-8.

McBride, C. (2000) *Dependently Typed Functional Programs and their Proofs*. PhD Thesis, University of Edinburgh.

McBride, C. (2002) Elimination with a motive. In *Types for Proofs and Programs*, Callaghan, P., Luo, Z., McKinna, J., & Pollack, R. (eds). Berlin, Heidelberg: Springer, pp. 197–216. ISBN 978-3-540-45842-5.

McBride, C., Goguen, H., & McKinna, J. (2006) A few constructions on constructors. In *Types for Proofs and Programs*, Filliâtre, J.-C., Paulin-Mohring, C., and Werner, B. (eds). Berlin, Heidelberg: Springer, pp. 186–200. ISBN 978-3-540-31429-5.

McKinna, J. & Forsberg, F. N. (2015) *The Encode-Decode Method, Relationally*, Institute of Cybernetics at Tallinn University of Technology, pp. 63–64. ISBN 978-9949-430-86-4.

Miquel, A. (2010) Re: Agda with the excluded middle is inconsistent?. Accessed April 17, 2017. Available at: `https://lists.chalmers.se/pipermail/agda/2010/001543.html` (Proof posted by Chung-Kil Hur on the Agda mailing list).

Monin, J.-F. (2010) Proof Trick: Small Inversions. In *Second Coq Workshop*, Bertot, Y. (ed). Edinburgh, U.K. Accessed April 17, 2017. Available at: `https://hal.inria.fr/inria-00489412`.

Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD Thesis, Chalmers University of Technology.

Peebles, D. (2012) Case splitting emits hidden record patterns that should remain implicit. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/635` (on the Agda bug tracker).

Reed, J. (2009) Higher-order constraint simplification in dependent type theory. In Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice. ACM, pp. 49–56.

Sicard-Ramírez, A. (2016) The `--without-K` option generates unsolved metas. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1775` (on the Agda bug tracker).

Sozeau, M. (2010) Equations: A dependent pattern-matching compiler. In *Interactive Theorem Proving*, Kaufmann, M. & Paulson, L. C. (eds). Berlin, Heidelberg: Springer, pp. 419–434. ISBN 978-3-642-14052-5.

Streicher, T. (1993) Investigations into intensional type theory. Habilitation thesis, Ludwig Maximilian University of Munich.

The Univalent Foundations Program. (2013) Homotopy Type Theory: Univalent Foundations of Mathematics. Accessed April 17, 2017. Available at: `http://homotopytypetheory.org/book`, Institute for Advanced Study.

Vezzosi, A. (2015) Heterogeneous equality incompatible with univalence even `--without-k`. Accessed April 17, 2017. Available at: `https://github.com/agda/agda/issues/1408` (on the Agda bug tracker).

Ziliani, B. & Sozeau, M. (2015) A unification algorithm for Coq featuring universe polymorphism and overloading. In International Conference on Functional Programming, ICFP.

Appendix A. Incompatibility of Injective Type Constructors with Univalence and Excluded Middle

The proofs of these two theorems are not essential for the understanding of the work in this paper. However, to our knowledge, there is no easy reference for them and we think they are interesting enough to mention here.

Theorem 92. *MLTT extended with univalence and injective type constructors is inconsistent.*

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

Theorem 93. *MLTT extended with the excluded middle and injective type constructors is inconsistent.*

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

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

We have

$$\mathtt{E}\ (\mathtt{D}\ F) \equiv_{\mathtt{Set}\to\mathtt{Set}} F \tag{A.1}$$

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

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

To come to the contradiction, consider the term $\mathtt{B} = \mathtt{E}\ (\mathtt{D}\ \mathtt{C})\ (\mathtt{D}\ \mathtt{C})$. By (A.1), we have $\mathtt{B} \equiv_{\mathtt{Set}} \mathtt{C}\ (\mathtt{D}\ \mathtt{C})$. Is $\mathtt{B}$ equal to $\bot$ or not? By the excluded middle, there are two cases:

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

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

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