Lifting Proof-Relevant Unification to Higher Dimensions

Jesper Cockx Dominique Devriese
iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium.
firstname.lastname@cs.kuleuven.be

Abstract

In a dependently typed language such as Coq or Agda, unification can be used to discharge equality constraints and detect impossible cases automatically. By nature of dependent types, it is necessary to use a *proof-relevant* unification algorithm where unification rules are functions manipulating equality proofs. This ensures their correctness but simultaneously sets a high bar for new unification rules. In particular, so far no-one has given a satisfactory proof-relevant version of the injectivity rule for indexed datatypes.

In this paper, we describe a general technique for solving equations between constructors of indexed datatypes. We handle the main technical problem—generalization over equality proofs in the indices—by introducing new equations between equality proofs. Borrowing terminology from homotopy type theory, we call them *higher-dimensional equations*. To apply existing one-dimensional unifiers to these higher-dimensional equations, we show how unifiers can be lifted to a higher dimension. We show the usefulness of this idea by applying it to the unification algorithm used by Agda, though it can also be applied in languages that support identity types but not general indexed datatypes.

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

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

1. Introduction

When writing programs or proofs in a dependently typed language, you often encounter equality proofs in the context that you'd like to discharge. For example, doing a case analysis on a vector of type $\texttt{Vec}\ A\ m$ produces two cases: one for the empty vector where $m = \texttt{zero}$ and one for prepending an element to a vector where $m = \texttt{suc}\ n$ for some n. Usually, these equations are not very interesting and you want to solve them as soon as possible.

However, simply substituting by these equations is not always sufficient because other terms and their types may depend on the *proofs* of these equations. For example, when you want to prove some property $P\ v$ of a vector $v : \texttt{Vec}\ A\ m$ and you are in the case of an empty vector, we cannot use $P\ []$ as the goal type since this is ill-typed: P takes an argument of type $\texttt{Vec}\ A\ m$, not $\texttt{Vec}\ A\ \texttt{zero}$. Instead, the equality proof $e : \texttt{zero} \equiv_{\mathbb{N}} m$ is needed to construct the goal type $P\ (\texttt{subst}\ (\texttt{Vec}\ A)\ e\ [])$.

To make progress on a large class of problems like this, you can use a proof-relevant unification algorithm as we proposed in previous work (Cockx et al. 2016). In this framework, unification problems are represented as telescopes containing flexible variables and equations, and unification rules are type-theoretic *equivalences* between two such telescopes.[1] For example, the injectivity rule for the $\texttt{suc}$ constructor for natural numbers is represented by an equivalence of type $(e : \texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n) \simeq (e : m \equiv_{\mathbb{N}} n)$. This means any use of possible axioms is made explicit in the construction of the equivalence. Another advantage of this approach is that the equivalence describes exactly how a proof of $\texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n$ can be transformed into a proof of $m \equiv_{\mathbb{N}} n$ and vice versa, i.e. it has a certain *computational content*. This is useful for the translation of pattern matching to eliminators (Goguen et al. 2006; Cockx et al. 2014).

Another feature of unification in a dependently typed setting is that the type of an equation can depend on the proof of earlier equations. This is necessary when you encounter equations between dependently typed terms. For example in the unification problem $(e_1 : m \equiv_{\mathbb{N}} n)(e_2 : v \equiv_{\texttt{Vec}\ A\ e_1} w)$ the two vectors $v : \texttt{Vec}\ A\ m$ and $w : \texttt{Vec}\ A\ n$ have a different type but this poses no problem because we have a proof $e_1 : m \equiv_{\mathbb{N}} n$ (see Side note 1 for an explanation of our notation for heterogeneous equalities).

Keeping track of the dependencies between equations is important for dealing with equations between construc-

[1] As standard in homotopy type theory, an equivalence consists of a function between the telescopes together with a left and a right inverse.

tors of an indexed datatype such as $\texttt{Vec}$. For example, if you have two equations $(e_1 : \texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n)(e_2 : \texttt{cons}\ m\ x\ xs \equiv_{\texttt{Vec}\ A\ e_1} \texttt{cons}\ n\ y\ ys)$ then you can simplify both equations at once by applying the injectivity rule for $\texttt{cons}$, resulting in the new set of equations $(e'_1 : m \equiv_{\mathbb{N}} n)(e'_2 : x \equiv_A y)(e'_3 : xs \equiv_{\texttt{Vec}\ A\ e'_1} ys)$.

In general, the injectivity rule is an equivalence:

$$(\bar{e} : \bar{\imath} \equiv_\Phi \bar{\jmath})(e : \texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{e}} \texttt{c}\ \bar{v}) \simeq (\bar{e}' : \bar{u} \equiv_\Delta \bar{v}) \quad (1)$$

i.e. it can be applied to an equation between two equal constructors where the type is a datatype applied to distinct equality proofs for its indices (Cockx et al. 2016). In this case we say that the indices are *fully general*.

Problem statement. The main question posed in this paper is what you can do if you encounter an equation of the form $\texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{\imath}} \texttt{c}\ \bar{v}$ but the indices $\bar{\imath}$ are not fully general. For example, you may encounter an equation:

$$(e : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ n)} \texttt{cons}\ n\ y\ ys) \quad (2)$$

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. However, we will see that it is possible to construct an equivalence between this equation and $(e'_1 : x \equiv_A y)(e'_2 : xs \equiv_{\texttt{Vec}\ A\ n} ys)$, so there is no fundamental reason why unification should fail. See also issue #1775 reported by Sicard-Ramírez (2016) on the Agda bug tracker for another instance of this problem.

In 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 unification rules to homogeneous equations and additionally imposed a *self-unifiability criterion* to the indices of the datatype when applying the injectivity rule. In practice, this meant that the injectivity rule could only be applied when the indices consisted of closed constructor forms only (e.g. $\texttt{suc}\ (\texttt{suc}\ \texttt{zero})$), a severe restriction to the applicability of the rule. In Cockx et al. (2016) we used the general (heterogeneous) version of the injectivity rule and relied on *reverse unification* to generalize the indices. This method had some potential in theory, but turned out to be too difficult to implement in practice. Neither did it take into account the type of the constructor in question, so they didn't include useful heuristics such as forced constructor arguments (Brady et al. 2003).

Contributions.

- We show how the injectivity rule for indexed datatypes can be made more generally applicable by generalizing over the indices in the type of the equation, generating higher-dimensional equations in the process.
- We prove that all regular unification rules can also be applied to higher-dimensional equations by *lifting* them to these higher dimensions.
- We explain how higher-dimensional unification formalizes the concept of forced constructor arguments, a heuristic that allows unification to skip certain constructor arguments if they are determined by the type of the constructor.
- We describe our implementation of higher-dimensional unification used in the implementation of the Agda language, showing that the algorithm also works in practice.

Overview. Section 2 describes higher-dimensional unification, first by means of an example and then in full generality. Section 3 contains the theoretical details that show how higher-dimensional unification works behind the scenes. Section 4 contains a more in-depth look at some interesting aspects of higher-dimensional unification, and Section 5 discusses the implementation of our ideas in the Agda language. Finally, Section 6 contains a few pointers to related work and Section 7 concludes.

2. Higher-dimensional unification

The theory of higher-dimensional unification presented in this paper builds on our previous work on proof-relevant unification (Cockx et al. 2016), so we start with a quick overview of the contents of that paper. After that, we explain the idea of higher-dimensional unification on an example before moving on to the general case.

2.1 Unifiers as Equivalences

Unification problems. Unification problems are represented by telescopes of the form $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ where Γ contains the flexible variables, Δ is the telescope of equations and $\bar{u}$ and $\bar{v}$ are the left- and right-hand sides of the equations respectively. Here, the telescopic identity type $\bar{e} : \bar{u} \equiv_\Delta \bar{v}$ is defined inductively on the length of the telescope by:

$$\begin{array}{lll} () \equiv_{()} () & := & () \\ (e;\bar{e} : s;\bar{s} \equiv_{(x:A)\Delta} t;\bar{t}) & := & (e : s \equiv_A t) \\ & & (\bar{e} : \bar{s} \equiv_{\Delta[x\mapsto e]} \bar{t}) \end{array} \quad (3)$$

where we write $()$ for both the empty telescope and the empty list of terms. The meaning of the heterogeneous equality $\bar{s} \equiv_{\Delta[x\mapsto e]} \bar{t}$ is further explained in Side note 1.

Unifiers. A unifier for the unification problem $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ is represented by a reduced telescope Γ' together with a telescope map $f : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$, i.e. a function returning values for the variables in Γ plus proofs that the equations are satisfied for these values.

A unifier is a most general one if the function f is an equivalence $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \simeq \Gamma'$, i.e. if f has both a left and a right inverse.

The unification algorithm. 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 given in Figure 1 to the unification problem, simplifying one or more equations in

Side note 1: cubical syntax. In this paper we use an unconventional notation for equality proofs inspired by cubical type theory (Cohen et al. 2015). Concretely, this notation manifests itself in three places:

- If $E : A \equiv_{\mathtt{Set}_i} B$ is an equality proof between two types A and B and we have $x : A$ and $y : B$, then $x \equiv_E y$ stands for the heterogeneous equality $\mathtt{coerce}\ E\ x \equiv_B y$ where $\mathtt{coerce} : A \equiv_{\mathtt{Set}_i} B \to A \to B$.
- Each term doubles as its own proof of reflexivity, i.e. if $u : A$ then we write u instead of $\mathtt{refl}_u$ when giving a proof of $u \equiv_A u$.
- Each function doubles as its own proof of congruence, i.e. if $f : A \to B$ then we write $f\ e$ instead of $\mathtt{cong}\ f\ e$ when constructing a proof of $f\ u \equiv_B f\ v$ from a proof $e : u \equiv_A v$. Here, congruence $\mathtt{cong}$ (also called $\mathtt{ap}$ in HoTT circles) has type $(f : (x : A) \to B\ x)(e : u \equiv_A b) \to f\ x \equiv_{B\ e} f\ y$.

This notation allows us to define substitution of a variable x with an equality proof e as $u[x \mapsto e] := (\lambda x.\, u)\ e$. Note that $x \equiv_{P\ e} y$ is nothing but the type $\mathtt{PathOver}\ P\ e\ x\ y$ from The Univalent Foundations Program (2013).

each step. This process continues until one of three possible situations occurs:

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

2.2 Higher-dimensional unification by example

Before we describe higher-dimensional unification in full generality, we start with a motivating example. Suppose we are working on the unification problem:

$$\Gamma(e : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys) \qquad (4)$$

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

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

$$\begin{aligned}&\Gamma(e_1 : \mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n)\\ &(e_2 : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys)\\ &(p : e_1 \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})\end{aligned} \qquad (5)$$

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

Now we are free to apply the $\mathtt{injectivity}_{\mathtt{cons}}$ rule to e_2, resulting in the unification problem:

$$\begin{aligned}&\Gamma(e_1' : n \equiv_{\mathbb{N}} n)(e_2' : x \equiv_A y)(e_3' : xs \equiv_{\mathtt{Vec}\ A\ e_1'} ys)\\ &(p : \mathtt{suc}\ e_1' \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})\end{aligned} \qquad (6)$$

Note that applying the injectivity rule to e_2 has instantiated the variable e_2 with the expression $\mathtt{suc}\ e_1'$ in the type of p, which is cubical notation for $\mathtt{cong}\ \mathtt{suc}\ e_1'$. This instantiation is determined by the computational behavior of the $\mathtt{injectivity}_{\mathtt{suc}}$ rule (see Side note 2). As you can see, p has become 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, it is useful to first consider a one-dimensional version of this problem:

$$(e_1' : \mathbb{N})(e_2' : A)(e_3' : \mathtt{Vec}\ A\ e_1')(p : \mathtt{suc}\ e_1' \equiv_{\mathbb{N}} \mathtt{suc}\ n) \qquad (7)$$

We have kept the names e_1', e_2' and e_3' from (6) but they now represent regular variables instead of equations. This is a problem we know how to solve: we can apply $\mathtt{injectivity}$ and $\mathtt{solution}$ to find an equivalence f between this telescope and $(e_2' : A)(e_3' : \mathtt{Vec}\ A\ n)$. This solves the one-dimensional problem.

Step 3: lifting unifiers to a higher dimension. How does this help us with solving the higher-dimensional problem? By Theorem 2 (see Section 3), we can *lift* the equivalence f to get a new equivalence $f^{\uparrow}$:

$$\begin{aligned}&(e_1' : n \equiv_{\mathbb{N}} n)(e_2' : x \equiv_A y)(e_3' : xs \equiv_{\mathtt{Vec}\ A\ e_1'} ys)\\ &(p : \mathtt{suc}\ e_1' \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{suc}\ n)\\ &\quad\simeq\\ &(e_2'' : x \equiv_A y)(e_3'' : xs \equiv_{\mathtt{Vec}\ A\ n} ys)\end{aligned} \qquad (8)$$

This solves the higher-dimensional equation p, as well as the reflexive equation e_1' (we didn't have to use the fact that $\mathbb{N}$ satisfies uniqueness of identity proofs!).

Finally, we can apply the $\mathtt{solution}$ rule twice to solve the equations e_2'' and e_3'', setting $y := x$ and $ys := xs$. So putting everything together, we have found an equivalence between the original telescope (4) and $(n : \mathbb{N})(x : A)(xs : \mathtt{Vec}\ A\ n)$, solving the unification problem.

$$\begin{aligned}
\texttt{solution} &: (x : A)(e : x \equiv_A u) \simeq () && \text{(if } x \text{ doesn't occur freely in } u\text{)}\\
\texttt{deletion} &: (e : u \equiv_A u) \simeq () && \text{(if } A \text{ satisfies UIP)}\\
\texttt{injectivity}_{\texttt{c}} &: (\bar{k}[\Delta \mapsto \bar{u}]; \texttt{c}\ \bar{u} \equiv_{\texttt{D}}\ \bar{k}[\Delta \mapsto \bar{v}]; \texttt{c}\ \bar{v}) \simeq (\bar{u} \equiv_{\Delta} \bar{v}) && \text{(if } \texttt{c} : \Delta \to \texttt{D}\ \bar{k} \text{ is a constructor)}\\
\texttt{conflict}_{\texttt{c}_1,\texttt{c}_2} &: (\bar{k}_1[\Delta_1 \mapsto \bar{u}]; \texttt{c}_1\ \bar{u} \equiv_{\texttt{D}}\ \bar{k}_2[\Delta_2 \mapsto \bar{v}]; \texttt{c}_2\ \bar{v}) \simeq \bot && \text{(if } \texttt{c}_\texttt{i} : \Delta_i \to \texttt{D}\ \bar{u}_i \text{ are distinct constructors)}\\
\texttt{cycle}_{x,u} &: (\bar{k}; x \equiv_{\texttt{D}}\ \bar{l}; u) \simeq \bot && \text{(if } x \text{ occurs strongly rigid in } u\text{)}
\end{aligned}$$

Figure 1: The basic unification rules formulated as equivalences as given by Cockx et al. (2016). Here we only give the type of each rule, but we also need the computational behavior of each rule to know their effect on other equations. For example, because $\texttt{solution}^{-1}\ () = u; \texttt{refl}$ we know that `solution` assigns u to the variable x. The computational behavior of the other rules is given in Side note 2.

2.3 The general case

Now that we have seen how to solve the problem in an example, let's try to generalize the solution. We follow the same three steps as in the example, so if anything is unclear you can jump back to the corresponding step in the example.

In the general situation, we are trying to solve a unification problem of the form:

$$\Gamma(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(e : \texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{\imath}}\ \texttt{c}\ \bar{v})(\bar{e}_2 : \bar{u}_2 \equiv_{\Delta_2} \bar{v}_2) \quad (9)$$

where $\texttt{c} : \Delta_{\texttt{c}} \to \texttt{D}\ \bar{k}$ is a constructor of the indexed datatype $\texttt{D} : \Phi \to \texttt{Set}_i$. Note that $\bar{\imath}$ are not just terms of type Φ but *proofs* that the indices in the type of $\texttt{c}\ \bar{u}$ and $\texttt{c}\ \bar{v}$ are equal, i.e. $\bar{\imath} : \bar{k}[\Delta_{\texttt{c}} \mapsto \bar{u}] \equiv_{\Phi} \bar{k}[\Delta_{\texttt{c}} \mapsto \bar{v}]$. From here on, we write $\bar{k}_u$ for $\bar{k}[\Delta_{\texttt{c}} \mapsto \bar{u}]$ and $\bar{k}_v$ for $\bar{k}[\Delta_{\texttt{c}} \mapsto \bar{v}]$.

Step 1: generalizing the indices. If $\bar{\imath}$ is fully general (i.e. consists of distinct variables from $\bar{e}_1$), then we can apply the $\texttt{injectivity}_{\texttt{c}}$ rule to simplify the equation e. Otherwise, we first need to generalize the indices in order to proceed.

For now, we will focus on just the equation $\bar{e}_1$ and e, the rest of the telescope (Γ and $\bar{e}_2$) will be added again in Step 3. To generalize the indices $\bar{\jmath}$ in the type of e, we introduce new variables $\bar{\jmath} : \bar{k}_u \equiv_{\Phi} \bar{k}_v$ together with equalities $\bar{p} : \bar{\imath} \equiv_{\bar{k}_u \equiv_{\Phi} \bar{k}_v} \bar{\jmath}$:[2]

$$\begin{array}{l}
(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(e : \texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{\imath}}\ \texttt{c}\ \bar{v})\\
\quad \simeq\\
(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(\bar{\jmath} : \bar{k}_u \equiv_{\Phi} \bar{k}_v)\\
(e : \texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{\jmath}}\ \texttt{c}\ \bar{v})(\bar{p} : \bar{\imath} \equiv_{\bar{k}_u \equiv_{\Phi} \bar{k}_v} \bar{\jmath})
\end{array} \quad (10)$$

[2] This is the exact same technique as used by McBride (1998): to do a case split on a variable $x : \texttt{D}\ \bar{\imath}$ where $\bar{\imath}$ is not fully general, he introduces new variables $\bar{\jmath} : \Phi$ together with equalities $\bar{p} : \bar{\imath} \equiv_{\Phi} \bar{\jmath}$. This means that now $x : \texttt{D}\ \bar{\jmath}$ where $\bar{\jmath}$ are just variables, so it is possible to perform a case split on x. The only difference in our case is that we are working one dimension higher, i.e. we work with equations between elements of the datatype instead of elements of the datatype itself.

Since $\bar{\jmath}$ consists of distinct variables, it's now possible to apply $\texttt{injectivity}_{\texttt{c}}$ to the equation e. This gives us an equivalence:

$$\begin{array}{l}
(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(\bar{\jmath} : \bar{k}_u \equiv_{\Phi} \bar{k}_v)\\
(e : \texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{\jmath}}\ \texttt{c}\ \bar{v})(\bar{p} : \bar{\imath} \equiv_{\bar{k}_u \equiv_{\Phi} \bar{k}_v} \bar{\jmath})\\
\quad \simeq\\
(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(\bar{e}' : \bar{u} \equiv_{\Delta_{\texttt{c}}} \bar{v})(\bar{p} : \bar{\imath} \equiv_{\bar{k}_u \equiv_{\Phi} \bar{k}_v} \bar{k}_e)
\end{array} \quad (11)$$

where $\bar{k}_e = \bar{k}[\Delta_{\texttt{c}} \mapsto \bar{e}']$. Note in particular that we now have a non-trivial higher-dimensional unification problem $\bar{p}$.

Step 2: lowering the dimension of equations. At first sight, it would seem that we need an entirely new set of unification rules to solve higher-dimensional unification problems (except for the solution rule, which can be used at any dimension). But as we will see, it is possible to reuse the existing unification rules on higher-dimensional problems. In fact, our notation for equality proofs is already strongly suggestive of how this works: 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 $\texttt{suc}\ e_1 \equiv_{\texttt{suc}\ x \equiv_{\mathbb{N}} \texttt{suc}\ y} \texttt{suc}\ e_2$ to $e_1 \equiv_{x \equiv_{\mathbb{N}} y} e_2$.

In general, whenever we encounter a higher-dimensional unification problem $\bar{u} \equiv_{\bar{x} \equiv_{\Delta} \bar{y}} \bar{v}$ we lower it by one dimension and consider the problem $\bar{u} \equiv_{\Delta} \bar{v}$ where the equation variables in $\bar{u}$ and $\bar{v}$ are treated as regular variables. If we manage to find a solution to this one-dimensional problem, we can then *lift* this solution to the higher dimension. The technical result that makes this possible is Theorem 2 in the next section.

In the problem we have on hand now, this means we consider the one-dimensional unification problem $(\bar{e}_1 : \Delta_1)(\bar{e}' : \Delta_{\texttt{c}})(\bar{p} : \bar{\imath} \equiv_{\Phi} \bar{k}_e)$. Note that $\bar{e}_1$ are now regular variables of type Δ_1, so the $\bar{\imath}$ in this equation are normal terms rather than equality proofs. Since this is a one-dimensional unification problem, we can apply the known unification rules from Figure 1 to solve it. For what follows, we assume that we can find a solution to this problem in the form of an equivalence:

$$f : (\bar{e}_1 : \Delta_1)(\bar{e}' : \Delta_{\texttt{c}})(\bar{p} : \bar{\imath} \equiv_{\Phi} \bar{k}_e) \simeq \Delta_1' \quad (12)$$

Side note 2: computational behavior of unifiers. The computational behavior of the unifier f suddenly becomes very relevant for the type of the resulting unification problem! In particular, we need to determine the behavior of the functions $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to \Gamma'$ and $f^{-1} : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$. Here we give the computational behavior for the unification rules in Figure 1:

- All the rules preserve the invariant that $\Gamma' \subseteq \Gamma$, so $f : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to \Gamma'$ picks out the variables from Γ that also occur in Γ'.
- The rule $\mathtt{solution}^{-1} : () \to (x : A)(e : x \equiv_A t)$ returns t for the variable x and $\mathtt{refl}$ for the proof e, and $\mathtt{deletion}^{-1} : () \to (e : t \equiv_A t)$ returns $\mathtt{refl}$. Meanwhile, $\mathtt{injectivity_c}^{-1} : (\bar{x} \equiv_\Delta \bar{y}) \to (\bar{u}[\Delta \mapsto \bar{x}]; \mathtt{c}\ \bar{x} \equiv_{\hat{\mathtt{D}}} \bar{u}[\Delta \mapsto \bar{y}]; \mathtt{c}\ \bar{y})$ maps proofs $\bar{e} : \bar{x} \equiv_\Delta \bar{y}$ to $\bar{u}[\Delta \mapsto \bar{e}]; \mathtt{c}\ \bar{e}$.

Step 3: lifting unifiers to a higher dimension. Now we have to lift this solution back to the higher-dimensional problem. Theorem 2 gives us a lifted equivalence $f^{\uparrow}$:

$$\begin{array}{l}(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(\bar{e}' : \bar{u} \equiv_{\Delta_{\mathtt{c}}} \bar{v})(\bar{p} : \bar{\imath} \equiv_{\bar{k}_u \equiv_\Phi \bar{k}_v} \bar{k}_e) \\ \quad \simeq \\ (\bar{e}'_1 : f\ \bar{u}_1\ \bar{u}\ \overline{\mathtt{refl}} \equiv_{\Delta'_1} f\ \bar{v}_1\ \bar{v}\ \overline{\mathtt{refl}})\end{array} \tag{13}$$

This is exactly what we need to solve the problem in (11). To calculate the left- and right-hand sides of the equations $\bar{e}'_1$, we need to know the computational content of the unifier f as detailed in Side note 2.

Now we can combine the equivalences in (10), (11) and (13), to get the equivalence:

$$\begin{array}{l}(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(e : \mathtt{c}\ \bar{u} \equiv_{\mathtt{D}\ \bar{\imath}} \mathtt{c}\ \bar{v}) \\ \quad \simeq \\ (\bar{e}'_1 : f\ \bar{u}_1\ \bar{u}\ \overline{\mathtt{refl}} \equiv_{\Delta'_1} f\ \bar{v}_1\ \bar{v}\ \overline{\mathtt{refl}})\end{array} \tag{14}$$

To get a solution to the full unification problem in (9), we have to add Γ and $(\bar{e}_2 : \bar{u}_2 \equiv_{\Delta_2} \bar{v}_2)$ to both sides of the equivalence. This gives us the final equivalence:

$$\begin{array}{l}\Gamma(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(e : \mathtt{c}\ \bar{u} \equiv_{\mathtt{D}\ \bar{\imath}} \mathtt{c}\ \bar{v})(\bar{e}_2 : \bar{u}_2 \equiv_{\Delta_2} \bar{v}_2) \\ \quad \simeq \\ \Gamma(\bar{e}'_1 : f\ \bar{u}_1\ \bar{u}\ \overline{\mathtt{refl}} \equiv_{\Delta'_1} f\ \bar{v}_1\ \bar{v}\ \overline{\mathtt{refl}})(\bar{e}_2 : \bar{u}_2 \equiv_{\Delta'_2} \bar{v}_2)\end{array} \tag{15}$$

Since Δ_2 is dependent on $\bar{e}_1$ and e, it has to be updated on the right-hand side of the equivalence by applying the substitution (14). Specifically, Δ'_2 is obtained by substituting $(f^{-1}\ \bar{e}'_1)|_{\bar{e}_1}$ for $\bar{e}_1$ and $\mathtt{c}\ (f^{-1}\ \bar{e}'_1)|_{\bar{e}'}$ for e in Δ_2.[3] Again we need the computational content of the unifier f, this time of the function f^{-1}. This is also described in Side note 2.

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.

[3] We write $\cdot|_{\bar{e}_1}$ for the projection from the telescope onto the values of $\bar{e}_1$, and similarly for $\cdot|_{\bar{e}'}$.

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

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

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

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

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

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

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

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

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

+ reflexivity, symmetry, transitivity and congruence rules for =

Figure 2: The core formal rules of UTT, including dependent function types $(x : A) \to B$, an infinite hierarchy of universes $\mathtt{Set}_0$ $(= \mathtt{Set})$, $\mathtt{Set}_1$, $\mathtt{Set}_2$,…, and β-equality.

3. Theoretical details

In the last section, we have seen how higher-dimensional unification can be applied to make the injectivity rule more generally applicable. In this section, we dive into the heart of the problem: how can we lift unifiers (i.e. equivalences) to a higher dimension? Our main result is Theorem 2, telling us exactly how to update the left- and right-hand sides of the equations when lifting a unifier.

Our setting. We work in a fairly minimal version of intentional type theory based on UTT (Luo 1994), see Figure 2 for the basic rules of this theory. Additionally, we make use of Martin-Löf (1984)'s identity type $x \equiv_A y$ and Dybjer (1991)'s indexed datatypes $\mathtt{D} : \Phi \to \mathtt{Set}_i$ defined by a number of constructors $\mathtt{c_i} : \Delta_i \to \mathtt{D}\ \bar{u}_i$.

All other constructions we use can be defined in terms of this basic theory:

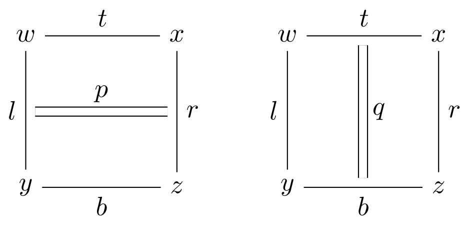
- A telescope $(x : A)(y : B)(z : C)$ is interpreted as an iterated sigma type $\Sigma_{(x:A)}\ (\Sigma_{(y:B)}\ C)$. This allows us to use the regular function type also for maps between telescopes.
- An equivalence of type $A \simeq B$ is a tuple of a function $f : A \to B$ together with left and right inverses $g_1, g_2 : B \to A$ plus proofs that they are actually inverses.

In particular, we do not make use of any additional type-theoretic axioms such as uniqueness of identity proofs or univalence.

Squares. To structure our reasoning about higher-dimensional equalities, we make use of the concept of a *square*, also called a *2-path* by The Univalent Foundations Program (2013):

Definition 1 (Square)**.** Let $A : \texttt{Set}_i$, $w, x, y, z : A$, $t : w \equiv_A x$, $b : y \equiv_A z$, $l : w \equiv_A y$, and $r : x \equiv_A z$. The *square type* $\texttt{Square}\ t\ b\ l\ r$ is defined to be the dependent equality type $l \equiv_{t \equiv_A b} r$.

If we imagine a square with top side t, bottom side b, left side l, and right side r, then $\texttt{Square}\ t\ b\ l\ r$ can be thought of as the type of identity proofs that fill this square horizontally as visualized in Figure 3a.



(a) Horizontal filling $p : \texttt{Square}\ t\ b\ l\ r$ (b) Vertical filling $q : \texttt{Square}\ l\ r\ t\ b$

Figure 3: The $\texttt{Square}$ type represents the possible ways to fill a square defined by four equality proofs.

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 3. We need to rely on the fact that both square types are in fact equivalent:

Lemma 1 (Flipping squares)**.** *Let $A : \texttt{Set}$, $w, x, y, z : A$, $t : w \equiv_A x$, $b : y \equiv_A z$, $l : w \equiv_A y$, and $r : x \equiv_A z$. Then we can construct an equivalence $\texttt{flip}\ t\ b\ l\ r : \texttt{Square}\ t\ b\ l\ r \simeq \texttt{Square}\ l\ r\ t\ b$.*

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

For the construction of the left and right inverse of $\texttt{flip}$, we can just change the order of t, b, l and r in the construction of $\texttt{flip}$. For the proofs that they are in fact inverses, the same sequence of path inductions as used in the construction of $\texttt{flip}$ suffices. □

Lifting unifiers. To lift an equivalence to a higher dimension, we make use of Theorem 2.11.1 from The Univalent Foundations Program (2013), which we repeat here:

Theorem 1 (Lifting of equivalences)**.** *If a function $f : A \to B$ is an equivalence and $x, y : A$, then $\texttt{cong}\ f : x \equiv_A y \to f\ x \equiv_A f\ y$ is also an equivalence.*

This theorem can already take us *almost* all the way to lifting a unifier, with the missing piece turning out to be exactly Lemma 1. This realization allows us to prove our main theorem:

Theorem 2 (Lifting of unifiers)**.** *Suppose we have a unifier $f : \Delta(\bar{p} : \bar{a} \equiv_\Phi \bar{b}) \simeq \Delta'$ and terms $\bar{u}, \bar{v} : \Delta$ together with proofs $\bar{r} : \bar{a}[\Delta \mapsto \bar{u}] \equiv_{\Phi[\Delta \mapsto \bar{u}]} \bar{b}[\Delta \mapsto \bar{u}]$ and $\bar{s} : \bar{a}[\Delta \mapsto \bar{v}] \equiv_{\Phi[\Delta \mapsto \bar{v}]} \bar{b}[\Delta \mapsto \bar{v}]$. Then we can construct a lifted unifier $f^\uparrow$:*

$$\begin{aligned}&(\bar{e} : \bar{u} \equiv_\Delta \bar{v})(\bar{p} : \bar{a}[\Delta \mapsto \bar{e}] \equiv_{\bar{r} \equiv_\Phi \bar{s}} \bar{b}[\Delta \mapsto \bar{e}])\\ &\quad\simeq\\ &(e' : f\ \bar{u}\ \bar{r} \equiv_{\Delta'} f\ \bar{v}\ \bar{s})\end{aligned} \tag{16}$$

Proof. Applying Theorem 1 with $x = \bar{u}; \bar{r}$ and $y = \bar{v}; \bar{s}$ gives us an equivalence $\texttt{cong}\ f$:

$$\begin{aligned}&(\bar{e} : \bar{u} \equiv_\Delta \bar{v})(\bar{q} : \bar{r} \equiv_{\bar{a}[\Delta \mapsto \bar{e}] \equiv_\Phi \bar{b}[\Delta \mapsto \bar{e}]} \bar{s})\\ &\quad\simeq\\ &(\bar{e}' : f\ \bar{u}\ \bar{r} \equiv_{\Delta'} f\ \bar{v}\ \bar{s})\end{aligned} \tag{17}$$

This is already very close to the result we want: the only problem is that $\bar{q}$ doesn't have the same type as $\bar{p}$ in the theorem statement. But if we think of both types as a square, then we see that the only difference between the two types is the direction in which the square is filled. This is illustrated in Figure 4. Hence we can apply Lemma 1, taking $w = x = \bar{r}$, $y = z = \bar{s}$, $t = \bar{r}$, $b = \bar{s}$, $l = \bar{a}[\Delta \mapsto \bar{e}]$, and $r = \bar{b}[\Delta \mapsto \bar{e}]$. This gives us an equivalence:

$$\begin{aligned}&(\bar{p} : \bar{a}[\Delta \mapsto \bar{e}] \equiv_{\bar{r} \equiv_\Phi \bar{s}} \bar{b}[\Delta \mapsto \bar{e}])\\ &\quad\simeq\\ &(\bar{q} : \bar{r} \equiv_{\bar{a}[\Delta \mapsto \bar{e}] \equiv_\Phi \bar{b}[\Delta \mapsto \bar{e}]} \bar{s})\end{aligned} \tag{18}$$

Composing this equivalence with $\texttt{cong}\ f$ gives us the desired equivalence $f^\uparrow$. □

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

$$\begin{tikzcd}
\bar{a}[\Delta \mapsto \bar{u}] \arrow[r, dash, "\bar{r}"] \arrow[d, dash, "{\bar{a}[\Delta \mapsto \bar{e}]}"'] \arrow[rd, phantom, "\bar{p}"] & \bar{b}[\Delta \mapsto \bar{u}] \arrow[d, dash, "{\bar{b}[\Delta \mapsto \bar{e}]}"] \\
\bar{a}[\Delta \mapsto \bar{v}] \arrow[r, dash, "\bar{s}"'] & \bar{b}[\Delta \mapsto \bar{v}]
\end{tikzcd}$$

(a) Horizontal filling $\bar{p}$

$$\begin{tikzcd}
\bar{a}[\Delta \mapsto \bar{u}] \arrow[r, dash, "\bar{r}"] \arrow[d, dash, "{\bar{a}[\Delta \mapsto \bar{e}]}"'] \arrow[rd, phantom, "\bar{q}"] & \bar{b}[\Delta \mapsto \bar{u}] \arrow[d, dash, "{\bar{b}[\Delta \mapsto \bar{e}]}"] \\
\bar{a}[\Delta \mapsto \bar{v}] \arrow[r, dash, "\bar{s}"'] & \bar{b}[\Delta \mapsto \bar{v}]
\end{tikzcd}$$

(b) Vertical filling $\bar{q}$

Figure 4: To construct the equivalence in Theorem 2, we need to apply Lemma 1 to transform the horizontal filling $\bar{p}$ into a vertical one $\bar{q}$.

4. Discussion

In this section, we highlight three features of higher-dimensional unification: its relation with forcing rules, the observation that it cannot end in a conflict, and its applicability in languages that don't support general indexed datatypes.

Forcing rules. Forcing (Brady et al. 2003) is a compiler optimization for dependently typed programming languages that erases constructor arguments that are fully determined by the type of the constructor. For example, the argument $(n : \mathbb{N})$ of the `cons` constructor of `Vec` can be erased because it is fully determined by the type of the constructor.

During unification, it is intuitively clear that forced arguments can safely be skipped. This intuition can be formally justified by using higher-dimensional unification. For example, for any proof $u : m \equiv_{\mathbb{N}} n$, higher-dimensional unification gives us:

$$\begin{array}{l}
(e : \texttt{cons}\ m\ x\ xs \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ u)} \texttt{cons}\ n\ y\ ys) \\
\quad \simeq \\
(e_1 : m \equiv_{\mathbb{N}} n)(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\texttt{Vec}\ A\ e_1} ys) \\
(p : \texttt{suc}\ e_1 \equiv_{\texttt{suc}\ m \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{suc}\ u) \\
\quad \simeq \\
(e_1 : m \equiv_{\mathbb{N}} n)(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\texttt{Vec}\ A\ e_1} ys) \\
(p : e_1 \equiv_{m \equiv_{\mathbb{N}} n} u) \\
\quad \simeq \\
(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\texttt{Vec}\ A\ u} ys)
\end{array} \tag{19}$$

telling us exactly that we can skip unification of the forced argument of `cons`!

In this example, higher-dimensional unification formally justifies the heuristic that 'forced arguments need not be unified'. We believe that this observation also holds more generally, i.e. that higher-dimensional unification can replace the heuristic generally. Unfortunately, while there exists a formalisation of forcing for other reasons (memory usage during evaluation, simpler term equivalence checking), we have not found a formalisation of how precisely forcing can be applied during unification, so that we cannot be sure in general.

No conflict at higher dimensions. 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:

$$\texttt{cong}\ \texttt{c}_1\ \bar{e}_1 \equiv \texttt{cong}\ \texttt{c}_2\ \bar{e}_2 \tag{20}$$

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:

$$e \equiv \texttt{cong}\ \texttt{c}\ e \tag{21}$$

where the left-hand side has some type $u \equiv v$ but the right-hand side has type $\texttt{c}\ \bar{u}' \equiv \texttt{c}\ \bar{v}'$. As a consequence, we do not have to deal with the possibility of a negative success in our implementation of higher-dimensional unification.

Higher-dimensional unification in the absence of indexed datatypes. An interesting question is to what extent higher-dimensional unification is coupled to the use of indexed data types. In particular, would we also encounter higher-dimensional unification problems if we used constructors with embedded equality proofs instead of indices? This question is especially relevant since languages like Coq discourage the use of indexed datatypes. Datatypes with embedded equality proofs are also more suitable for languages that have canonical identity proofs other than `refl`, like HoTT and cubical type theory. In these languages, indexed datatypes can still be used but they are mere syntactic sugar for the actual datatypes with embedded equality proofs.

It turns out that the answer to this question is yes. We illustrate this by working out the example from Section 2 again for a version of the `Vec` datatype with embedded equality proofs instead of indices. Suppose `Vec` A n is defined with constructors `[]` $: n \equiv_{\mathbb{N}}$ `zero` $\to$ `Vec` $A\ n$ and `cons` $: (m : \mathbb{N})(x : A)(xs : \texttt{Vec}\ A\ m) \to n \equiv_{\mathbb{N}} \texttt{suc}\ m \to$ `Vec` $A\ n$ and consider the unification problem:

$$(e : \texttt{cons}\ k\ x\ xs\ \texttt{refl} \equiv_{\texttt{Vec}\ A\ (\texttt{suc}\ k)} \texttt{cons}\ k\ y\ ys\ \texttt{refl}) \tag{22}$$

Since this version of the `Vec` datatype doesn't have an index, we can apply the $\texttt{injectivity}_{\texttt{cons}}$ rule to simplify this equation to:

$$\begin{array}{l}
(e_1 : k \equiv_{\mathbb{N}} k)(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\texttt{Vec}\ A\ e_1} ys) \\
(e_4 : \texttt{refl} \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ e_1} \texttt{refl})
\end{array} \tag{23}$$

Now e_4 is an equation between equality proofs, much like the one we obtained in (6), 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 1 shows that these two types are in fact equivalent. So we can conclude that 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.

5. Implementation

We implemented the ideas presented in this paper as an extension to the unification algorithm used by Agda for checking definitions by dependent pattern matching (Cockx et al. 2016). This addition allows Agda to typecheck more definitions by pattern matching, such as the example given by Sicard-Ramírez (2016) on the Agda bug tracker.

Our implementation closely follows the steps from Section 2.3. In particular, when applying the injectivity rule to a unification problem of the form $\Gamma(\bar{e}_1 : \bar{u}_1 \equiv_{\Delta_1} \bar{v}_1)(e : \texttt{c}\ \bar{u} \equiv_{\texttt{D}\ \bar{\imath}} \bar{v})(\bar{e}_2 : \bar{u}_2 \equiv_{\Delta_2} \bar{v}_2)$ the unification algorithm constructs the new unification problem $\Delta_1\Delta_{\texttt{c}}(\bar{p} : \bar{\imath} \equiv_{\Phi} \bar{k})$ and recursively calls itself on this new problem.

One noteworthy fact about the implementation is how the left- and right-hand sides $f\ \bar{u}_1\ \bar{u}\ \overline{\texttt{refl}}$ and $f\ \bar{v}_1\ \bar{v}\ \overline{\texttt{refl}}$ of the new unification problem in (15) are computed. The implementation doesn't have an explicit representation of the function f, so it's not possible to calculate them directly. Instead, the recursive call produces a substitution ρ of type $\Delta'_1 \to \Delta_1\Delta_{\texttt{c}}$. This allows us to calculate $f^{-1} : \Delta'_1 \to (\bar{e}_1 : \Delta_1)(\bar{e}' : \Delta_{\texttt{c}})(\bar{p} : \bar{\imath} \equiv_{\Phi} \bar{k}_e)$ as $\lambda \bar{x}'_1.\ \bar{x}'_1\rho; \overline{\texttt{refl}}$, but doesn't give us a direct way to compute f.

To go in the opposite direction, we view ρ as a *pattern* with free variables from Δ'_1. This allows us to match the values from $\Delta_1\Delta_{\texttt{c}}$ against this pattern. The proofs of $\bar{\imath} \equiv_{\Phi} \bar{k}_e$ (assumed to be $\overline{\texttt{refl}}$ in our implementation) ensure that this matching cannot fail, so this allows us to recover the values of the variables in Δ'_1, thus computing the function $f : (\bar{e}_1 : \Delta_1)(\bar{e}' : \Delta_{\texttt{c}})(\bar{p} : \bar{\imath} \equiv_{\Phi} \bar{k}_e) \to \Delta'_1$.

6. Related work

The contents of this paper build on previous work on proof-relevant unification by McBride (1998, 2000, 2002) and Goguen et al. (2006), as well as our own previous work (Cockx et al. 2014, 2016). Compared to the reverse unification rules in Cockx et al. (2016), higher-dimensional unification takes information into account from the types of the constructors as well as the types of the equation. This difference is very similar to the inversion of an inductive hypothesis by using a diagonalizer (Cornes and Terrasse 1995) versus using unification for the problem (McBride 1998).

The idea to view equality proofs themselves as the subjects of unification is inspired by cubical type theory (Cohen et al. 2015), where equality proofs are terms viewed 'one level up'. 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.

7. Conclusion

The literature on dependent types is often focused on how they can be used to rule out bad programs, but what's equally important is how types can guide your development in the right direction. In the case of unification, the syntactic injectivity had to be ruled out when taking a proof-relevant perspective on unification. But actually, a previously invisible structure was waiting to be discovered in the types of the equations. In this paper, we show how to exploit this higher-dimensional structure to increase the power of unification.

The addition of higher-dimensional structure to the unification process only provides a real benefit when one takes dependently typed programming seriously: it overcomes the gap between unification of simply-typed terms and dependently typed ones. By offloading the mechanical part of reasoning about equality proofs to the unifier, you can use more sophisticated dependently typed data structures without paying for it with increased complexity. We hope this brings the day that programming and proving with dependent types become as commonplace as regular static typing now one step closer.

References

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for proofs and programs*, 2003.

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

Jesper Cockx, Dominique Devriese, and Frank Piessens. Unifiers as Equivalences: proof-relevant unification of dependently typed data. In *Proceedings of the 21th ACM SIGPLAN international conference on Functional programming*. ACM, 2016.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom, 2015. Preprint.

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

Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Proceedings of the first workshop on Logical frameworks*, 1991.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*. 2006.

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

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

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

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

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

Andrés Sicard-Ramírez. The –without-K option generates unsolved metas, 2016. URL `https://github.com/agda/agda/issues/1775`. On the Agda bug tracker.

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