*JFP* **26**, e16, 40 pages, 2016.  
doi:10.1017/S0956796816000174

# *Eliminating dependent pattern matching without K*

JESPER COCKX, DOMINIQUE DEVRIESE
and FRANK PIESSENS
*iMinds-Distrinet, KU Leuven*
(*e-mail:* jesper.cockx@cs.kuleuven.be, dominique.devriese@cs.kuleuven.be, frank.piessens@cs.kuleuven.be)

**Abstract**

Dependent pattern matching is an intuitive way to write programs and proofs in dependently typed languages. It is reminiscent of both pattern matching in functional languages and case analysis in on-paper mathematics. However, in general, it is incompatible with new type theories such as homotopy type theory (HoTT). As a consequence, proofs in such theories are typically harder to write and to understand. The source of this incompatibility is the reliance of dependent pattern matching on the so-called K axiom – also known as the uniqueness of identity proofs – which is inadmissible in HoTT. In this paper, we propose a new criterion for dependent pattern matching without K, and prove it correct by a translation to eliminators in the style of Goguen *et al.* (2006 *Algebra, Meaning, and Computation*). Our criterion is both less restrictive than existing proposals, and solves a previously undetected problem in the old criterion offered by Agda. It has been implemented in Agda and is the first to be supported by a formal proof. Thus, it brings the benefits of dependent pattern matching to contexts where we cannot assume K, such as HoTT.

## 1 Introduction

Dependent pattern matching (Coquand, 1992) is a technique for writing functions in languages based on dependent type theory, such as Agda (Norell, 2007), Coq (Sozeau, 2010), and Idris (Brady, 2013). It allows us to define functions in a style similar to functional programming languages such as Haskell, by giving a number of equalities called *clauses*. For example, the function $\mathtt{half} : \mathbb{N} \to \mathbb{N}$ can be defined as

$$\begin{array}{llcl} \mathtt{half} : & \mathbb{N} & \to & \mathbb{N} \\ \mathtt{half} & \mathtt{zero} & = & \mathtt{zero} \\ \mathtt{half} & (\mathtt{suc}\ \mathtt{zero}) & = & \mathtt{zero} \\ \mathtt{half} & (\mathtt{suc}\ (\mathtt{suc}\ k)) & = & \mathtt{suc}\ (\mathtt{half}\ k) \end{array} \tag{1}$$

Note that pattern matching combines two powerful programming features, namely *case analysis* and *recursion*.

Additionally, dependent pattern matching can be used to write *proofs* (in the form of dependently typed functions). For example, we can prove the transitivity of the propositional equality $x \equiv y$ (Martin-Löf, 1984; Paulin-Mohring, 1993) by pattern

matching on its only constructor $\mathtt{refl}$ of type $x \equiv x$:

$$\begin{aligned} &\mathtt{trans} : (x\ y\ z : A) \to x \equiv y \to y \equiv z \to x \equiv z \\ &\mathtt{trans}\ x\ \lfloor x \rfloor\ \lfloor x \rfloor\ \mathtt{refl}\ \mathtt{refl} = \mathtt{refl} \end{aligned} \tag{2}$$

*Inaccessible patterns*, like $\lfloor x \rfloor$ in this example, witness the fact that only one type-correct argument can be in that position. Indeed, matching on a proof of $x \equiv y$ with $\mathtt{refl} : x \equiv x$ forces $x$ and $y$ to be the same. Another example is the proof $\mathtt{cong}$ that any function maps equal arguments to equal results:

$$\begin{aligned} &\mathtt{cong} : (f : A \to B)(x\ y : A) \to x \equiv y \to f\ x \equiv f\ y \\ &\mathtt{cong}\ f\ x\ \lfloor x \rfloor\ \mathtt{refl} = \mathtt{refl} \end{aligned} \tag{3}$$

Proofs by dependent pattern matching are typically much shorter and more readable than ones that use the classical *datatype eliminators* associated with each inductive family (see next section for more on eliminators). For example, let $\leqslant$ be the usual ordering on $\mathbb{N}$ defined as an inductive family (Dybjer, 1991) with two constructors $\mathtt{lz}$ and $\mathtt{ls}$:

$$\begin{aligned} &\mathtt{lz} : (n : \mathbb{N}) \to \mathtt{zero} \leqslant n \\ &\mathtt{ls} : (m\ n : \mathbb{N}) \to m \leqslant n \to \mathtt{suc}\ m \leqslant \mathtt{suc}\ n \end{aligned} \tag{4}$$

We can prove antisymmetry of this relation by pattern matching as follows:

$$\begin{aligned} &\mathtt{antisym} : (m\ n : \mathbb{N}) \to m \leqslant n \to n \leqslant m \to m \equiv n \\ &\mathtt{antisym}\ \lfloor \mathtt{zero} \rfloor\ \lfloor \mathtt{zero} \rfloor\ (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor)\ (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor) = \mathtt{refl} \\ &\mathtt{antisym}\ \lfloor \mathtt{suc}\ m \rfloor\ \lfloor \mathtt{suc}\ n \rfloor\ (\mathtt{ls}\ m\ n\ x)\ (\mathtt{ls}\ \lfloor n \rfloor\ \lfloor m \rfloor\ y) = \mathtt{cong}\ \mathtt{suc}\ (\mathtt{antisym}\ m\ n\ x\ y) \end{aligned} \tag{5}$$

Pattern matching allows us to skip the two cases where one of the arguments is $\mathtt{lz}\ n$ and the other is $\mathtt{ls}\ n'\ m'$ because $\mathtt{zero}$ can never be of the form $\mathtt{suc}\ m'$ (this is called the **conflict** rule). In the second clause, $m'$ (the first argument of the second $\mathtt{ls}$) was replaced by $\lfloor n \rfloor$ because $\mathtt{suc}\ m'$ and $\mathtt{suc}\ n$ were forced to be equal, and similarly $n'$ (its second argument) is replaced by $\lfloor m \rfloor$ (this is called the **injectivity** rule).

**Desugaring pattern matching.** In a dependent type theory with inductive families but without pattern matching, functions have to be written using datatype eliminators. For example, the standard eliminator for the $\leqslant$ is

$$\begin{aligned} \mathtt{elim}_{\leqslant} : (P &: (m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to \mathtt{Set}_i) \to \\ &(m_{\mathtt{lz}} : (n : \mathbb{N}) \to P\ \mathtt{zero}\ n\ (\mathtt{lz}\ n)) \to \\ &(m_{\mathtt{ls}} : (m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to \\ &\qquad P\ m\ n\ x \to P\ (\mathtt{suc}\ m)\ (\mathtt{suc}\ n)\ (\mathtt{ls}\ m\ n\ x)) \to \\ &(m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to P\ m\ n\ x \end{aligned} \tag{6}$$

Here, $P$ is called the *motive* (McBride, 2002) of the eliminator and $m_{\mathtt{lz}}$ and $m_{\mathtt{ls}}$ are called the *methods*. The eliminator $\mathtt{elim}_{\leqslant}$ has the following evaluation rules:

$$\begin{aligned} \mathtt{elim}_{\leqslant}\ P\ m_{\mathtt{lz}}\ m_{\mathtt{ls}}\ \mathtt{zero}\ n\ (\mathtt{lz}\ n) &= m_{\mathtt{lz}}\ n \\ \mathtt{elim}_{\leqslant}\ P\ m_{\mathtt{lz}}\ m_{\mathtt{ls}}\ (\mathtt{suc}\ m)\ (\mathtt{suc}\ n)\ (\mathtt{ls}\ m\ n) &= m_{\mathtt{ls}}\ m\ n \end{aligned} \tag{7}$$

Eliminators will be defined in general in Section 4.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

```
antisym : (m n : ℕ) → m ≤ n → n ≤ m → m ≡ n
antisym = elim≤ (λm; n; _. n ≤ m → m ≡ n)
  (λn; e. elim≤ (λn; m; _. m ≡ zero → m ≡ n)
    (λn; e. e)
    (λk; l; _; _; e. elim⊥ (λ_. suc l ≡ suc k) (noConfℕ (suc l) zero e))
    n zero e refl)
  (λm; n; _; H; q. cong suc (H
    (elim≤ (λk; l; _. k ≡ suc n → l ≡ suc m → n ≤ m)
      (λ_; e; _. elim⊥ (λ_. n ≤ m) (noConfℕ zero (suc n) e))
      (λk; l; e; _; x; y. subst (λn. n ≤ m)
        (noConfℕ (suc k) (suc n) x)
        (subst (λm. k ≤ m) (noConfℕ (suc l) (suc m) y) e))
      (suc n) (suc m) y refl refl)))
```

Fig. 1. This proof of the antisymmetry of $\leqslant$ is more complex than the proof by pattern matching (5) because it uses only the standard datatype eliminators (see Section 4) and the "no confusion" property of the natural numbers. No confusion can be constructed from the eliminator for $\mathbb{N}$ as well (see Section 5), but expanding this construction here would make the proof even more complex.

Figure 1 gives an alternative definition of `antisym` that only uses eliminators. All the equational reasoning that was done automatically in the definition by pattern matching now has to be done explicitly. The proof with eliminators also requires considerable work for the construction of the motive of each eliminator, whilst this can be done automatically in many cases, including definitions by pattern matching (McBride, 2002). So it is clearly preferable to use pattern matching for this proof.

As shown by Goguen *et al.* (2006), *all* definitions by dependent pattern matching can be translated to ones that only use eliminators. However, for this translation they depend on the so-called K axiom. Coquand (1992) already observed that pattern matching allows proving this K axiom[1]:

$$\begin{aligned} &\mathrm{K} : (P : a \equiv a \to \mathtt{Set})(p : P\ \mathtt{refl})(e : a \equiv a) \to P\ e \\ &\mathrm{K}\ P\ p\ \mathtt{refl} = p \end{aligned} \tag{8}$$

The K axiom is equivalent with the *uniqueness of identity proofs* principle, which states that any two proofs of $x \equiv y$ must be equal. As observed by Hofmann and Streicher (1994), the K axiom does not follow from the standard rules of type theory, but it is compatible with them.

So far, none of the examples we gave needs the K axiom for the translation to eliminators (except for the definition of K itself). The reason we need the K axiom is to deal with reflexive equations; for example, an equation $\mathtt{Bool} \equiv \mathtt{Bool}$. Remember that in type theory there is no strict boundary between types and terms, so we can form equations between types as well, as in this example. Given such an equation

[1] Actually, K is only an axiom in settings where we are unable to define it. Once we give a computational behaviour to K like we do here, it ceases to be an axiom and becomes a theorem instead. However, we will keep calling it the K axiom for the sake of tradition.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

between types, we can coerce terms of the first type to the other using the function $\texttt{coerce} : A \equiv B \to A \to B$ (which can be constructed by pattern matching on the proof of $A \equiv B$). Now, we can use pattern matching to prove that coercing `true` by any proof of $\texttt{Bool} \equiv \texttt{Bool}$ results in `true`:

$$\begin{aligned} &\texttt{coerce-id} : (e : \texttt{Bool} \equiv \texttt{Bool}) \to \texttt{coerce}\ e\ \texttt{true} \equiv \texttt{true} \\ &\texttt{coerce-id}\ \texttt{refl} = \texttt{refl} \end{aligned} \tag{9}$$

This can be desugared to

$$\texttt{coerce-id} = \lambda e.\ \mathrm{K}\ (\lambda e.\ \texttt{coerce}\ e\ \texttt{true} \equiv \texttt{true})\ \texttt{refl}\ e \tag{10}$$

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

**Pattern matching in HoTT.** An emerging field within dependent type theory is *homotopy type theory* (HoTT) (The Univalent Foundations Program, 2013). It gives a new interpretation of terms of type $x \equiv y$ as *paths* from $x$ to $y$. Many basic constructions in HoTT can be written very elegantly using pattern matching; for example, `trans` (2) corresponds to the composition of two paths, and `cong` (3) can be interpreted as a proof that all functions in HoTT are continuous, in the sense that they preserve paths.

One of the core elements of HoTT is the *univalence axiom*. This axiom states roughly that any two isomorphic types can be identified, i.e. if there is a function $f : A \to B$ which has both a left and a right inverse, then it gives us a proof $\texttt{ua}\ f$ of $A \equiv B$. Moreover, this proof satisfies $\texttt{coerce}\ (\texttt{ua}\ f)\ x = f\ x$. Univalence captures the common mathematical practice of informal reasoning "up to isomorphism" in a nice and formal way. It also has a number of useful consequences, such as *functional extensionality*.[2]

However, the univalence axiom is *incompatible* with dependent pattern matching. For example, we can construct a function $\texttt{swap} : \texttt{Bool} \to \texttt{Bool}$ such that $\texttt{swap}\ \texttt{true} = \texttt{false}$ and vice versa. This function is its own inverse, so by univalence it gives us a proof $\texttt{ua}\ \texttt{swap}$ of $\texttt{Bool} \equiv \texttt{Bool}$ such that coercing `true` along this proof results in `false`. Together with the proof `coerce-id` (9), this leads to a proof of the absurdity $\texttt{true} \equiv \texttt{false}$. This general incompatibility has forced people working on HoTT to avoid using pattern matching or risk unsoundness.

The source of the incompatibility between univalence and dependent pattern matching is that pattern matching relies on the K axiom. Intuitively, this makes sense because HoTT (and the univalence axiom in particular) encodes important information in equality proofs, whilst K is exactly the assertion that there is no such information. So if we want to be able to use dependent pattern matching in HoTT, we need a theory that keeps track of the information inside equality proofs, instead of discarding it like Goguen *et al.* (2006).

[2] There are other theories that also support functional extensionality, but in HoTT it can be proven using nothing more than univalence as ascribed to Voevodsky by The Univalent Foundations Program Section 4.9.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

**Avoiding K.** If we could somehow restrict definitions by pattern matching so that we could translate them to type theory with eliminators but without the K axiom, then we would be able to use pattern matching in HoTT. A previous attempt to achieve this was an option in Agda called `--without-K` (Norell *et al.* 2012). Before Agda version 2.4.0, it attempted to detect definitions by pattern matching that make use of the K axiom by means of a syntactic check. The goal of this option was to allow people to use pattern matching in a safe way when it is undesirable to assume K. However, the option has been criticized many times, for being too restrictive (Sicard-Ramírez, 2013), for having unclear semantics (Reed, 2013), and for containing errors (Altenkirch, 2012; Cockx, 2014). These errors allowed one to prove (weaker versions of) the K axiom. Whilst these errors are typically fixed quickly after being found, this situation really calls for a more in-depth investigation of dependent pattern matching without K.

**Contributions.**

- We present a new criterion that describes what kind of definitions by pattern matching are still allowed if we do not assume K. This criterion is strictly more general than previous attempts. In contrast to previous attempts, our criterion is not based on a syntactic check, but on a modification to the unification algorithm used for pattern matching.
- We give a formal proof that definitions by pattern matching satisfying this criterion are conservative over standard type theory by translating them to eliminators in the style of Goguen *et al.* (2006), but *without* relying on the K axiom. For this proof, we develop some tools for working with *homogeneous telescopic equality*. We also give generalized versions of the unification transitions used in Goguen *et al.* (2006), where the return type can depend on the equality proofs.
- Our criterion has been implemented as a patch to Agda. We test it on a body of examples in order to show its adequacy, soundness, and generality. As of Agda version 2.4.0 (released on June 5, 2014), our implementation replaces the old version of `--without-K`.
- Finally, we give a general way to detect specific datatypes that do satisfy the K axiom. This can be used to make our criterion even less restrictive.

An earlier version of this paper has appeared at the ICFP 2014 conference (Cockx *et al.*, 2014). This journal version extends the conference version by a discussion of the interaction between termination checking and the K axiom (Section 3.3) and a criterion to detect datatypes that satisfy K (Section 8). It also adds a number of examples, and compares our work with a new version of the Equations package for Coq (Sozeau, 2015) and the Lean theorem prover (de Moura *et al.*, 2015), neither of which were published at the time of writing the conference version.

**Overview.** The rest of this paper is organized as follows. In Section 2, we give the basic theory behind dependent pattern matching, and in Section 3, we describe our

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

$$\underline{n}\begin{cases}\texttt{zero} \mapsto \texttt{zero} \\ (\texttt{suc}\ \underline{m})\begin{cases}\texttt{suc}\ \texttt{zero} \mapsto \texttt{zero} \\ \texttt{suc}\ (\texttt{suc}\ k) \mapsto \texttt{suc}\ (\texttt{half}\ k)\end{cases}\end{cases}$$

Fig. 2. A representation of the function `half` by a case tree. At each internal node, the variable on which the case split is performed is underlined.

criterion that removes the dependence on the K axiom. Sections 4 through 7 contain the main technical contribution of this paper: a proof that definitions by pattern matching satisfying our criterion can be translated to eliminators without using K. Section 4 reviews the basics of type theory and inductive families, and Section 5 gives some general constructions that allow us to work with these inductive families. Section 6 makes use of these constructions to implement the unification transitions, the core element of the pattern matching translation. Finally, in Section 7, we bring all these elements together for the main proof. In Section 8, we discuss how to enhance our criterion with detection of types that satisfy K, and we discuss related work in Section 9 and future work in Section 10.

## 2 Dependent pattern matching: behind the scenes

To the user of a dependently typed language, a definition by pattern matching appears to be no more than a list of equations the function should satisfy. However, in order to translate such a list of clauses to a definition in terms of eliminators, they must first be translated to a number of consecutive case splits on the arguments. For example, the function $\texttt{half} : \mathbb{N} \to \mathbb{N}$ in Definition 1 is defined by first doing a case split on the argument $n : \mathbb{N}$ – giving us two cases $n = \texttt{zero}$ and $n = \texttt{suc}\ m$ – and then another case split on $m$.

Things get more complicated for an inductive family (Dybjer, 1991) such as $\texttt{Fin}\ n$, the canonical finite set of $n$ elements, or $m \leqslant n$, the type of proofs that $m$ is smaller than or equal to $n$. When splitting on a type from an inductive family, we need to apply *unification* in order to determine which constructors can occur in a given position.

In this section, we first describe how definitions by pattern matching can be represented as a *case tree* (Augustsson, 1985), where each node represents a case split. Next, we zoom in on the individual nodes, revealing how the subcases of each node are determined by a unification process.

### 2.1 Case trees

A definition by pattern matching consists of one or more case splits. We represent these case splits by a case tree. The nodes of a case tree for a function `f` are labelled by patterns, where the label of the root node consists of variables only. Each internal node of a case tree corresponds to a case split, whilst each leaf node corresponds to a clause of the definition. An example of a case tree for the function `half` (1) is given in Figure 2.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

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

Fig. 3. The structural order $\prec$ is used to check termination (Goguen *et al.*, 2006).

To construct a case tree from a given set of clauses, in each node one pattern variable is chosen on which to split the pattern. This variable must be a *blocking variable*: In at least one of the function clauses, there has to be a constructor pattern in the position of this variable. For each constructor of the correct type, one subtree is created where the variable has been replaced by this constructor applied to fresh variables. This process is repeated until there are no more blocking variables, at which point the leaf node is filled in by the right-hand side of the corresponding function clause.

Using case trees has a number of advantages. First, the patterns at the leaves of a case tree always form a covering, hence a representation as a case tree guarantees completeness of the definition. Second, they give an efficient method to evaluate functions defined by pattern matching (Maranget, 2008). Third and most importantly for our purposes, each internal node in a case tree corresponds exactly to the application of an eliminator for an inductive family, so constructing a case tree is a useful first step in the translation of dependent pattern matching to pure type theory as demonstrated by Goguen *et al.* (2006).

### 2.2 Structural recursion

In order to guarantee termination, functions are required to be *structurally recursive*. This means that the arguments of recursive calls should be *structurally smaller* than the pattern on the left-hand side. The structural order $\prec$ is defined in Figure 3. For functions with multiple arguments, the function should be structurally recursive on one of its arguments, i.e. there should be some $k$ such that $s_k \prec p_k$ for each clause $\mathtt{f}\ \bar{p} = t$ and each recursive call $\mathtt{f}\ \bar{s}$ in $t$.

### 2.3 Unification of the indices

When checking a definition that pattern matches on an element of an inductive family, we must decide which constructors can be used to construct a term of a particular type, and under which constraints. For example, consider the inductive family $m \leqslant n$ with constructors `lz` and `ls` as given in Definition 4. Suppose we want to do a case split on a variable of type $n \leqslant \mathtt{zero}$ as in the definition of `antisym` (5), then we have to decide for what arguments the two constructors produce a result of the form $n \leqslant \mathtt{zero}$. To do this, we try to unify the indices in the type of the variable with the indices of each constructor.

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

$$\begin{array}{rcl} n & \equiv_{\mathbb{N}} & \mathtt{zero}, \\ \mathtt{zero} & \equiv_{\mathbb{N}} & n' \end{array} \xrightarrow{m:=\mathtt{zero}} \mathtt{zero} \equiv_{\mathbb{N}} n \xrightarrow{n':=\mathtt{zero}} () \tag{11}$$

  Here, we renamed the argument $n$ of `lz` to $n'$ to avoid a name conflict.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

$$m\ n\ \underline{x}\ y \begin{cases} \lfloor \mathtt{zero} \rfloor\ n\ (\mathtt{lz}\ \lfloor n \rfloor)\ \underline{y} \{\ \lfloor \mathtt{zero} \rfloor\ \lfloor \mathtt{zero} \rfloor\ (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor)\ (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor) \mapsto \mathtt{refl} \\ \lfloor \mathtt{suc}\ m \rfloor\ n\ (\mathtt{ls}\ m\ \lfloor n \rfloor\ x)\ \underline{y} \begin{cases} \lfloor \mathtt{suc}\ m \rfloor\ \lfloor \mathtt{suc}\ n \rfloor\ (\mathtt{ls}\ m\ n\ x)\ (\mathtt{ls}\ \lfloor n \rfloor\ \lfloor m \rfloor\ y) \\ \mapsto \mathtt{cong}\ \mathtt{suc}\ (\mathtt{antisym}\ m\ n\ x\ y) \end{cases} \end{cases}$$

Fig. 4. A representation of the function `antisym` (5) by a case tree. Whilst there are two subtrees for the case split on $x$, each split on $y$ only has a single subcase due to the constraints on the type of $y$.

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

$$\begin{array}{rcl} n & \equiv_{\mathbb{N}} & \mathtt{suc}\ m', \\ \mathtt{zero} & \equiv_{\mathbb{N}} & \mathtt{suc}\ n' \end{array} \xrightarrow{n := \mathtt{suc}\ m'} \mathtt{zero} \equiv_{\mathbb{N}} \mathtt{suc}\ n \xrightarrow{\textbf{conflict}} \bot \tag{12}$$

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

This is reflected in the upper subtree of `antisym`'s case tree given in Figure 4, where there is only a case for $y = \mathtt{lz}\ \lfloor \mathtt{zero} \rfloor$, and none for $y = \mathtt{ls}\ m$. Similarly, for the second subtree, the indices are only has a case for the `ls` constructor, not for `lz`.

In general, suppose we are case splitting on a variable $x : \mathtt{D}\ \bar{u}$, where `D` is an inductive family with indices $\bar{u}$ (we consider `D` to already be applied to its parameters, if any). Suppose `D` has constructors $\mathtt{c}_i$ with return type $\mathtt{D}\ \bar{v}_i$ for $i = 1, \ldots, k$, then we have to *unify* $\bar{u}$ with each of the $\bar{v}_i$. Unification is the process of searching for *unifiers*, i.e. substitutions $\sigma$ such that $\bar{u}\sigma = \bar{v}_i\sigma$. A unification problem is represented as a list of equations $\Theta = (u_1 = v_{i,1}, \ldots, u_n = v_{in})$, and the following five unification transitions are used to simplify the problem step by step:

**Deletion:** $x = x, \Theta \Rightarrow \Theta$ (remove a reflexive equation from the list)
**Solution:** $x = t, \Theta \Rightarrow \Theta[x \mapsto t]$ (assign a value to the variable $x$ if $x$ is not free in $t$)
**Injectivity:** $\mathtt{c}\ \bar{s} = \mathtt{c}\ \bar{t}, \Theta \Rightarrow \bar{s} = \bar{t}, \Theta$ (applications of equal constructors can only be equal if their arguments are equal)
**Conflict:** $\mathtt{c}_1\ \bar{s} = \mathtt{c}_2\ \bar{t}, \Theta \Rightarrow \bot$ (applications of distinct constructors can never be equal)
**Cycle:** $x = \mathtt{c}\ \bar{p}[x], \Theta \Rightarrow \bot$ (a term can never be structurally smaller than itself)

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

**Positive success:** All equations have been solved, yielding a most general unifier $\sigma$.
**Negative success:** Either the **conflict** or the **cycle** rule applies, meaning that there exist no unifiers, i.e. this case is absurd.
**Failure:** An equation is reached for which no transition applies, meaning that the problem is too hard to be solved (by this unification algorithm).

This algorithm is complete for *constructor forms*: If both $\bar{u}$ and $\bar{v}$ are built from constructors and variables only, then unification will never result in a failure.

Case splitting succeeds if unification of $\bar{u}$ with each of the $\bar{v}_i$ succeeds (either positively or negatively). If all of them succeed negatively, we replace $x$ by an *absurd*

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

*pattern* $\emptyset$, marking that case splitting resulted in zero cases.[3] If on the other hand at least one of them succeeds positively, we get the same number of new cases where $x$ has been replaced by $\mathtt{c_i}\ \bar{y}$ and $\bar{y} : \Delta_i$ are fresh variables. To each of these cases, we then apply the substitution $\sigma_i$ constructed by unification. For example, a function $f : (n : \mathbb{N}) \to n \leqslant n \to P\ n$ can be defined by the following patterns:

$$\begin{array}{lcl} f\ \lfloor\mathtt{zero}\rfloor\ (\mathtt{lz}\ \lfloor\mathtt{zero}\rfloor) & = & [\ldots \text{ of type } P\ \mathit{zero}] \\ f\ \lfloor\mathtt{suc}\ n\rfloor\ (\mathtt{ls}\ n\ \lfloor n\rfloor\ x) & = & [\ldots \text{ of type } P\ (\mathtt{suc}\ n) \text{ where } n : \mathbb{N} \text{ and } x : n \leqslant n] \end{array} \tag{13}$$

Here, $\lfloor\ldots\rfloor$ marks an inaccessible pattern: It is not part of a case split, but rather computed by unification. The substitution $\sigma_i$ is also applied to the result type: In the first clause, the right-hand side should have type $P\ \mathtt{zero}$, whilst in the second one, it should have type $P\ (\mathtt{suc}\ n)$.

If the splitting done at each node of a case tree can be computed by the unification algorithm above and moreover the definition is structurally recursive, we call the case tree *valid*. If a case tree is valid, then Goguen *et al.* (2006) show that the function can be translated to one using eliminators and the K axiom.

## 3 A criterion for pattern matching without K

For function definition that match only on simple types, like the function `half` (1), each case split corresponds exactly to one application of the standard eliminator for $\mathbb{N}$, hence the K axiom is not needed. However, the unification algorithm used for case splitting on an inductive family depends crucially on the K axiom, so we have to restrict it in order to remove this dependence.

In this section, we describe our restricted unification algorithm that does not depend on K. We give a high-level view of the soundness proof of our criterion, which is the main subject of the rest of this paper. We also compare our criterion with the previous (syntactic) criterion for pattern matching without K in Agda. Finally, we give a short evaluation of our implementation of this criterion in Agda.

### *3.1 Restricting the unification rules*

Our criterion for pattern matching without K limits the unification algorithm in two ways:

- It is not allowed to use the **deletion** step.
- When applying the **injectivity** step on the equation $c\ \bar{s} = c\ \bar{t}$, where $c\ \bar{s}, c\ \bar{t} : D\ \bar{u}$, the indices $\bar{u}$ should be *self-unifiable*, i.e. unification of $\bar{u}$ with itself should succeed positively (whilst still adhering to these two restrictions).

This inevitably means that unification will fail more often. However, if unification results in a success (a positive or negative one), then we know that the original rules would have given the same result. Where the original algorithm was complete

[3] The reason for replacing $x$ by an absurd pattern instead of removing the pattern entirely, is to keep coverage checking decidable (Goguen *et al.*, 2006).

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

for constructor forms, our modified version is only complete for *linear* constructor forms (i.e. ones where each variable occurs only once).

### 3.2 Examples and counterexamples

As a first example, our criterion allows the definition of the J-eliminator for propositional equality[4] by pattern matching:

$$\begin{aligned} &\mathrm{J} : (P : (b : A) \to a \equiv b \to \mathtt{Set})(p : P\ a\ \mathtt{refl})(b : A)(e : a \equiv b) \to P\ b\ e \\ &\mathrm{J}\ \ P\ p\ \lfloor a \rfloor\ \mathtt{refl} = p \end{aligned} \tag{14}$$

Note that J does not express the property that the only equality proofs are given by `refl` , that property is expressed by K (Licata, 2011).

The unification problem for the case split on $e : a \equiv b$ with the constructor $\mathtt{refl} : a \equiv a$ is given by $b = a$. Unification succeeds positively after one **solution** step, with the most general unifier $[b \mapsto a]$ as the result. Likewise, the definitions of `trans` (2), `cong` (3), and `antisym` (5) in the introduction are also accepted.

In contrast, the definition of K by pattern matching is not allowed, as case splitting on the argument of type $a \equiv a$ produces a unification problem $a = a$, which fails without the **deletion** step of the unification algorithm.

$$\begin{aligned} &\mathrm{K} : (P : a \equiv a \to \mathtt{Set})(p : P\ \mathtt{refl})(e : a \equiv a) \to P\ e \\ &\mathrm{K}\ P\ p\ \mathtt{refl} = p \end{aligned} \tag{15}$$

This already explains the need for the first restriction to the unification algorithm.

As an example of why the second restriction is needed, consider the following weaker variant of K:

$$\begin{aligned} &\mathtt{weakK} : (P : \mathtt{refl} \equiv_{a \equiv a} \mathtt{refl} \to \mathtt{Set}) \to \\ &\qquad (p : P\ \mathtt{refl})(e : \mathtt{refl} \equiv_{a \equiv a} \mathtt{refl}) \to P\ e \\ &\mathtt{weakK}\ \ P\ p\ \mathtt{refl} = p \end{aligned} \tag{16}$$

The type of `weakK` says basically that any proof $e$ of $\mathtt{refl} \equiv \mathtt{refl}$ (where both instances of `refl` have type $a \equiv a$) must be equal to `refl`.[5] Like the regular K, this `weakK` does not follow from the standard rules of type theory and is incompatible with univalence (Kraus and Sattler, 2015). Case splitting on the argument $e$ of type $\mathtt{refl} \equiv_{a \equiv a} \mathtt{refl}$ requires unification of $\mathtt{refl} : a \equiv a$ with $\mathtt{refl} : a \equiv a$. Before applying the injectivity rule, the unifier will first check whether the index $a$ can be unified with itself. However, this fails because it is not allowed to apply the deletion rule, hence the definition of `weakK` is not accepted. It would be accepted if we did not have the second restriction to the unification algorithm.

[4] Following Paulin-Mohring (1993), we consider the first argument $x$ of the identity type $x \equiv y$ to be a datatype parameter instead of an index. This means this version of J corresponds to the principle of *based path induction* in HoTT.

[5] In the HoTT interpretation of elements of the identity type as paths, this would mean that there are no non-trivial paths between paths, i.e. all spaces have a dimension of at most 1.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

```
iso : One ≡ (⊥ → One)
iso = ua wrap
noo : (X : Set) → One ≡ X → X → ⊥
noo ⌊One⌋ refl (wrap f) = noo (One → ⊥) iso f
absurd : ⊥
absurd = noo (One → ⊥) iso (elim⊥ One)
```

Fig. 5. An example of what can go wrong when recursion on an argument of variable type is allowed. Here, `One : Set` is a datatype with a single constructor `wrap : (⊥ → One) → One`.

### *3.3 Interaction with termination checking*

One important but easily overlooked detail in the translation of dependent pattern matching to eliminators by Goguen *et al.* (2006) is that the type of the argument on which the function is structurally recursive must be a datatype. When working in a theory without the K axiom, this restriction becomes very important. Figure 5 gives an example[6] of what can go wrong if we would allow recursion on an argument of variable type.

In the example, `One : Set` is a datatype with a single constructor `wrap : (⊥ → One) → One`. Since `wrap` is a constructor and hence injective, it gives rise to an equivalence between `One` and `⊥ → One`. By univalence, it follows that these two types are equal (`iso`). The function `noo` illustrates the problem at hand: first, it pattern matches on a proof of `One ≡ X`, forcing $X$ to be equal to `One`. Next, it proceeds by induction on its third argument, which first had type $X$ but now type `One`. According to the naive interpretation, the function `noo` is structurally recursive on its third argument. However, the type of this argument changes from `One` in the argument position to `⊥ → One` in the recursive call. In effect, the definition of `noo` first strips the `wrap` constructor from its third argument in order to fool the termination checker, only to apply it again via a backdoor using the equality `iso`.

This type of recursion is not allowed by the eliminator for the `One` type, and is in fact incompatible with univalence as made evident by the proof `absurd` of `⊥`. So we need to be careful to disallow this kind of recursion, both in our proof and in the implementation of our criterion.

### *3.4 Soundness*

We have seen that our criterion rules out a direct definition of `K` (15) or a weaker form of it (16). But how can we know for sure that pattern matching doesn't allow us to prove anything that we wouldn't be able to using only the basic rules of type theory without the K axiom? We should prove that any definition by pattern matching satisfying our criterion could just as well be written using eliminators, like

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

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

Goguen *et al.* (2006) did for pattern matching *with* K. Formally, we will prove the following theorem:

*Theorem 1*
Let `f` : $(\bar{t} : \Delta) \to T$ be a function given by a valid case tree, adhering to the two restrictions given in Section 3.1. Then, we can construct a term `f'` : $(\bar{t} : \Delta) \to T$ constructed from eliminators only. Moreover, define $\{e\}_{\texttt{f}}^{\texttt{f'}}$ by replacing all occurrences of `f` by `f'` in $e$. Then, `f'` satisfies `f'` $\bar{t} \rightsquigarrow^* \{u\}_{\texttt{f}}^{\texttt{f'}}$ whenever `f` $\bar{t} \rightsquigarrow u$, i.e. it has the same reduction behaviour as `f`.

We give a high-level overview of the proof in this section, with more details in the following sections.

Our proof mostly follows the translation from pattern matching to eliminators by Goguen *et al.* (2006). There is a reason why it is hard to see where exactly the K axiom is used in their work: They do not use the axiom directly, but instead depend on the heterogeneous propositional equality. Heterogeneous equality $x \cong y$ allows the formation of equalities between terms $x : A$ and $y : B$ of different types. However, the only constructor of this type is `refl` : $x \cong x$, requiring that the types are in fact the same. This heterogeneous equality is convenient for expressing equality between sequences of data in a given telescope, as the types of later terms in the sequence may differ. Unfortunately, the elimination rule for this heterogeneous equality proposed by McBride is equivalent with the K axiom (McBride, 2000). Heterogeneous equality (and its elimination rule) is used almost everywhere in the translation, making it impossible to see where the K axiom is really needed, and where it's merely used out of convenience. So instead we work with the *homogeneous* propositional equality and the standard `J` eliminator.

The general idea of the proof is as follows. First, the definition by pattern matching is translated to a case tree as explained in Section 2.1, keeping into consideration the restrictions to the unification algorithm given in Section 3.1. Each leaf node of the case tree corresponds to a clause `f` $\bar{p} = e$, i.e. it defines `f` on arguments that match the pattern $\bar{p}$, and each internal node corresponds to a case split of $\bar{p}$ on some variable $x$ : `D` $\bar{u}$ into patterns $\bar{p}_1, \ldots, \bar{p}_n$. If we can assemble the definitions of `f` $\bar{p}_1, \ldots,$ `f` $\bar{p}_n$ into a definition of `f` $\bar{p}$, then we can work backwards from the leaf nodes towards the root, ultimately obtaining a definition of `f` on arbitrary variables.

So how do we assemble the definitions of `f` $\bar{p}_1, \ldots,$ `f` $\bar{p}_n$ into a definition of `f` $\bar{p}$? This assembly proceeds in two steps. First, we apply a technique called *basic* $\texttt{case}_{\texttt{D}}$*-analysis at* $\bar{u}; x$. This splits the problem into one subproblem for each constructor $c_i$ of `D`, and gives us proofs of the equations $\bar{u} = \bar{v}_i$ and $x =$ `c` $\bar{y}$. The second step is to apply *specialization by unification*, simplifying these equations step by step. The unification transitions make sure that we do not have to fill in anything for a negative success. Finally, we fill in the translated definition of `f` $\bar{p}_i$ for each positive success.

In general, there can be recursive calls to the function `f` in each clause `f` $\bar{p} = e$. These recursive calls are required to be structurally recursive on some argument $x$ : `D` $\bar{u}$ of `f`. It is important for the proof that the top-level type of $x$ in the type $\Delta$ of $f$'s arguments is already a datatype, not just the type of $x$ in each of the clauses

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

separately (see Section 3.3). This allows us to use well-founded recursion on `D` to obtain an inductive hypothesis $H$, asserting that `f` is already defined on arguments structurally smaller than $x$. This inductive hypothesis is then used to replace the recursive calls to `f` in $e$.

The challenge is then to construct all these techniques (case analysis, specialization by unification, and structural recursion) as terms *internal to type theory*. Before we begin this construction, we repeat some standard definitions from type theory (Section 4), including telescopic equality. We then recall some standard equipment for inductive datatypes given by McBride *et al.* (2006): case analysis, structural recursion, no confusion, and acyclicity, of which the latter two are slightly adapted to take the additional dependencies on equality proofs into account (Section 5). This is a return to an early version of McBride (1998) that was still based on homogeneous equality. No confusion and acyclicity are subsequently used to construct the unification transitions as terms inside type theory (Section 6). Finally, all these tools are brought together for the translation of case trees to eliminators (Section 7).

### *3.5 Comparison with the syntactic criterion*

So far, the only credible proposal of a criterion for pattern matching without K was the syntactic criterion previously used by Agda. So how does our criterion compare to it? One reason to prefer our criterion is that it is more amenable to the correctness proof given in Section 7. But we should also compare their generality, i.e. what kind of definitions are still allowed by each. The criterion previously used in Agda for pattern matching without K is specified as follows: If the flag is activated, then Agda only accepts certain case-splits. If the type of the variable to be split is `D pars ixs`, where `D` is a data (or record) type, `pars` stands for the parameters, and `ixs` the indices, then the following requirements must be satisfied:

- The indices `ixs` must be applications of constructors (or literals) to distinct variables. Constructors are usually not applied to parameters, but for the purposes of this check constructor parameters are treated as other arguments.
- These distinct variables must not be free in `pars`.

This criterion implies that the **deletion** rule is never used during unification. To see why this is true, note that it guarantees that all unification problems generated by pattern matching are of the form $\bar{u} = \bar{v}_i$, where $\bar{u}$ consists of constructors applied to free variables and each variable occurs only once in $\bar{u}$. Moreover, since new constructors introduced by case splitting are applied to fresh variables, the variables in $\bar{u}$ are not free in $\bar{v}_i$. Both the **solution** and the **injectivity** step preserve these three properties, hence we will never reach an equation of the form $x = x$.

On the other hand, the syntactic criterion does not imply that the indices are self-unifiable when applying the **injectivity** rule. But this is actually a bug in the syntactic criterion, allowing one to prove a weaker version of the K axiom (Cockx, 2014), similar to the example `weakK` (16). So the fact that our criterion is more restrictive in this case is actually a good thing.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

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

Another advantage of our criterion is that unlike the syntactic criterion, it does not put any requirements on the datatype parameters. This is very useful when we need injectivity of a constructor of a parameterized datatype. For example, the syntactic criterion does not allow case splitting on an argument of type $x :: xs \equiv y :: ys$, where :: is the list constructor, since the type $A$ of $x$ and $y$ is a parameter and the constructor :: is considered to be applied to this parameter. Our criterion has no such problems. This is especially useful in Agda since module parameters are also considered to be parameters of the datatypes defined inside that module chapter 4. So with the syntactic criterion, moving a definition to another module can cause an error, but with our criterion this is no longer the case.

Unfortunately, our criterion still has some limitations. For example, when working with the $\leqslant$ relation on finite sets `Fin` $n$, we cannot pattern match on an argument of type $i \leqslant i$, where $i$ : `Fin` $n$. This is because unification gets stuck on the problem `fs` $n\ x = $ `fs` $n\ y$, where the **deletion** rule is needed to remove the equation $n = n$. However, this definition is also refused by the syntactic criterion. In Section 8, we discuss possible solutions to problems of this kind.

Another limitation arises when an equation cannot be solved right away, but must be postponed until later. As the types of later equations may depend on the solution of these postponed equations, this may cause the types of both sides of an equation to be different. The algorithm presented in this paper expects the types to be (definitionally) equal, so it cannot deal with postponed equations.

### *3.6 Implementation*

Our new criterion for pattern matching without K has been implemented as a patch to Agda, included as of version 2.4.0. This patch consists of three changes to the typechecker when the `--without-K` option is enabled:

- Whenever the unification algorithm used by the case splitter encounters a reflexive equation $t = t$, instead of deleting the equation Agda throws an error, notifying the user that K has been disabled.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

- Whenever the unification algorithm encounters an equation `c` $\bar{u} =$ `c` $\bar{v}$, where `c` is the constructor of some inductive family `D`, Agda first tries to unify the indices of `D` with themselves before continuing to unify $\bar{u}$ with $\bar{v}$.
- When checking termination of a function $f$ that is structurally recursive on its $k$th argument, Agda checks whether the type of the $k$th argument of $f$ is actually a datatype. If not, Agda considers the function to be non-terminating.

We used our patch with a number of Agda programs in order to test it for adequacy, soundness, and generality.

**Adequacy.** In order to test the adequacy of our approach, we tested it on a number of small examples that should be definable without K, such as the functions `half` (1), `trans` (2), `cong` (3), and `antisym` (5) from the introduction. We also tested it on a body of Agda code related to propositional equality and HoTT by Danielsson (2013), which was written with Agda's current `--without-K` flag in mind. All these examples are accepted without problems.

**Soundness.** To test the soundness of our criterion, we also tested it on a number of variations on the K axiom and weaker versions of it. For example, when we try to define K as in Definition 15, we get the following error message: "Cannot eliminate reflexive equation $x = x$ of type $A$ because K has been disabled (when checking that the pattern `refl` has type $x \equiv x$)". Pattern matching with `refl` on a proof of `Bool` $\equiv$ `Bool` is also prohibited by our check. Similarly, the elimination rule for heterogeneous equality given by McBride (2000) (which is equivalent with K) is rejected, as are the weaker versions of K given by Altenkirch (2012) and Cockx (2014).

**Generality.** Finally, to test the generality of our approach, we gave it some definitions that are rejected by Agda's syntactic criterion, but do not actually rely on the K axiom. For example, definitions involving case splitting on types such as $m \leqslant m$, $k \equiv l + m$, and $x \equiv f\ y$ are accepted.

## 4 Type theory

As our version of type theory, we use Luo's Unified Theory of Dependent Types with dependent products, inductive families, and universes (Luo, 1994). We omit the meta-level logical framework and the impredicative universe of propositions because they are not needed for our current work. The formal rules of the version of Unified Theory of Dependent Types we use are summarized in Figure 6.

**Contexts and telescopes.** We use Greek capitals $\Gamma, \Delta, \ldots$ for both contexts and telescopes, capitals $T, U, \ldots$ for types, and small letters $t, u, \ldots$ for terms. Telescopes can best be thought of as the *tail* of a context: They are typed relative to a context, and they grow to the left rather than to the right. Note that the empty telescope () is inhabited by the empty list ().

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

$+$ reflexivity, symmetry, transitivity and congruence rules for $=$

Fig. 6. The core formal rules of UTT, including dependent function types $(x : A) \to B$, an infinite hierarchy of universes $\mathtt{Set}_0, \mathtt{Set}_1, \mathtt{Set}_2, \ldots$, and $\beta$-equality.

A list of terms is indicated by a bar above the letter, for example $\bar{t}$. Telescopes can take the role of the type of such a list of terms (see Figure 7), so we can write for example $\bar{t} : \Gamma$, where $\Gamma = (m : \mathbb{N})(p : m \equiv \mathtt{zero})$ and $\bar{t} = \mathtt{zero}; \mathtt{refl}$. More precisely, this means we give a semantics $[\![\Gamma]\!]$ to a telescope $\Gamma$ as iterated sigma types as defined in Figure 7. For example, $\Gamma \vdash a; b; c : (x : A)(y : B\ x)(z : C\ x\ y)$ stands for

$$\Gamma \vdash a, (b, (c, \mathtt{tt})) : \Sigma\ A\ (\lambda x.\, \Sigma\ (B\ x)\ (\lambda y.\, \Sigma\ (C\ x\ y)\ (\lambda z.\, \top))) \tag{17}$$

which is equivalent with saying $a : A$, $b : B\ a$, and $c : C\ a\ b$.

**Substitutions.** The simultaneous substitution of the terms $\bar{t}$ for the variables in the telescope $\Delta$ is written as $[\Delta \mapsto \bar{t}]$. We denote substitutions by small Greek letters $\sigma, \tau, \ldots$ A substitution can also be seen as a function between telescopes, i.e. if $\Gamma \vdash \bar{t} : \Delta$, then we have $\sigma = [\Delta \mapsto \bar{t}] : \Gamma \to \Delta$. The identity substitution is written as $id : \Delta \to \Delta$ and the composition of two substitutions $\sigma : \Delta_2 \to \Delta_3$ and $\tau : \Delta_1 \to \Delta_2$ is written as $\sigma \circ \tau : \Delta_1 \to \Delta_3$.

**Definitional and propositional equality.** In (intensional) type theory, there are two distinct notions of equality. On the one hand, two terms $s$ and $t$ are *definitionally equal* (or *convertible*) if we can derive $\Gamma \vdash s = t : T$, i.e. if $s$ and $t$ are equal up to

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

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

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

$$\begin{array}{lcl} [\![()]\!] & = & \top \\ [\![(x : A)\Delta]\!] & = & \Sigma\ A\ (\lambda x.[\![\Delta]\!]) \end{array}$$

Fig. 7. The typing rules for telescopes and their semantics as iterated sigma types.

$\beta$-reduction. On the other hand, two terms $s$ and $t$ are *propositionally equal* if we can prove their equality, i.e. if we can give a term of type $s \equiv t$. Propositional equality was introduced by Martin-Löf (Martin-Löf, 1984). We follow the definition of Paulin-Mohring (Paulin-Mohring, 1993), as an inductive family with two parameters $A : \mathtt{Set}_i$ and $a : A$, one index $b : A$, and one constructor $\mathtt{refl} : a \equiv a$. The standard eliminator for this datatype is exactly the J rule (14). Substitution by a propositional equality $\mathtt{subst} : (P : A \to \mathtt{Set}_i) \to x \equiv y \to P\ x \to P\ y$ can readily be defined from J by dropping the dependence of $P$ on the equality proof in the type of J. In the style of HoTT, we will write $e_*$ for $\mathtt{subst}\ P\ e$ when $P$ is clear from the context.

**Telescopic equality.** We define telescopic equality $\bar{s} \equiv \bar{t}$ inductively on the length of the telescope as follows:

$$\begin{array}{rcl} () & \equiv & () \quad := \quad () \\ s;\bar{s} & \equiv & t;\bar{t} \quad := \quad (e : s \equiv t)(\bar{e} : e_*\ \bar{s} \equiv \bar{t}) \end{array} \tag{18}$$

Note that the substitution $e_*$ is needed to make the equation between $\bar{s}$ and $\bar{t}$ again homogeneous. Here, we consider $\bar{s}$ to be an element of the iterated sigma type as defined in Figure 7 for the purpose of applying the substitution $e_*$. So to be fully explicit, the $e_*$ in the definition stands for $\mathtt{subst}\ (\lambda x : A.[\![\Gamma]\!])\ e$, where $(x : A)\Gamma$ is the type of $s;\bar{s}$ and $t;\bar{t}$. Telescopic inequality is defined by $\bar{s} \not\equiv \bar{t} := \bar{s} \equiv \bar{t} \to \bot$. For each $\bar{t} : \Delta$, we define $\overline{\mathtt{refl}} : \bar{t} \equiv \bar{t}$ as $\mathtt{refl};\ldots;\mathtt{refl}$. We also have the telescopic eliminator

$$\bar{\mathtt{J}} : (P : (\bar{s} : \Delta) \to \bar{r} \equiv \bar{s} \to \mathtt{Set}_i) \to P\ \bar{r}\ \overline{\mathtt{refl}} \to (\bar{s} : \Delta) \to (\bar{e} : \bar{r} \equiv \bar{s}) \to P\ \bar{s}\ \bar{e} \tag{19}$$

It is defined by eliminating the equations $\bar{e}$ from left to right using J:

$$\begin{array}{llllll} \bar{\mathtt{J}} & P & p & () & () & = & p \\ \bar{\mathtt{J}} & P & p & (s;\bar{s}) & (e;\bar{e}) & = & \mathtt{J}\ (\lambda s; e.\ (\bar{s} : \Delta)(\bar{e} : \bar{r} \equiv \bar{s}) \to P\ (s;\bar{s})\ (e;\bar{e})) \\ & & & & & & (\lambda \bar{s}; \bar{e}.\ \bar{\mathtt{J}}\ (\lambda \bar{s}; \bar{e}.\ P\ (r;\bar{s})\ (\mathtt{refl};\bar{e}))\ p\ \bar{e}) \\ & & & & & & e\ \bar{s}\ \bar{e} \end{array} \tag{20}$$

Each elimination of an equation $e_i : r_i \equiv s_i$ fills in $\mathtt{refl}$ 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. Telescopic substitution $\overline{\mathtt{subst}}$ is defined by dropping the dependence of $P$ on $\bar{r} \equiv \bar{s}$ in the definition of $\bar{\mathtt{J}}$. Again, we write $\bar{e}_*$ for $\overline{\mathtt{subst}}\ P\ \bar{e}$ when $P$ is clear from the context.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

**Inductive families.** Inductive families (Dybjer, 1991) are (dependent) types inductively defined by a number of *constructors*, for example $\mathbb{N}$ is defined by the constructors $\texttt{zero} : \mathbb{N}$ and $\texttt{suc} : \mathbb{N} \to \mathbb{N}$. Inductive families can also have *parameters* and *indices*,[7] for example, $\texttt{Vec}\ A\ n$ is an inductive family with one parameter $A : \texttt{Set}$, one index $n : \mathbb{N}$, and two constructors $\texttt{nil} : \texttt{Vec}\ A\ \texttt{zero}$ and $\texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)$. Each inductive family comes equipped with a *datatype eliminator*, for example, the eliminator for $\mathbb{N}$ is

$$\begin{aligned}\texttt{elim}_{\mathbb{N}} : (P : \mathbb{N} \to \texttt{Set}_i) \to (m_{\texttt{zero}} : P\ \texttt{zero}) \to \\ (m_{\texttt{suc}} : (n : \mathbb{N}) \to P\ n \to P\ (\texttt{suc}\ n)) \to \\ (n : \mathbb{N}) \to P\ n\end{aligned} \tag{21}$$

In general, let $\texttt{D}$ be an inductive family. Since everything we do in this paper is parametric in the datatype parameters of $\texttt{D}$, we consider $\texttt{D}$ to be already applied to (arbitrary) parameters. So $\texttt{D}$ is defined by the telescope $\Xi$ of the indices and the constructors:

$$\texttt{c}_i : \Delta_i \to (\Phi_{i,1} \to \texttt{D}\ \bar{v}_{i,1}) \to \ldots \to (\Phi_{i,n_i} \to \texttt{D}\ \bar{v}_{i,n_i}) \to \texttt{D}\ \bar{u}_i \tag{22}$$

for $i = 1, \ldots, k$. We write $\bar{\texttt{D}}$ for the telescope $(\bar{u} : \Xi)(x : \texttt{D}\ \bar{u})$. The standard eliminator for $\texttt{D}$ has a type of the form

$$\begin{aligned}\texttt{elim}_{\texttt{D}} : (P : \bar{\texttt{D}} \to \texttt{Set}_i)(m_1 : \ldots) \ldots (m_k : \ldots) \to \\ (\bar{x} : \bar{\texttt{D}}) \to P\ \bar{x}\end{aligned} \tag{23}$$

where the types of $m_1, \ldots, m_k$ are given by

$$\begin{aligned}m_i : (\bar{t} : \Delta_i) \to \\ (x_1 : \Phi_{i,1} \to \texttt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \texttt{D}\ \bar{v}_{i,n_i}) \to \\ (h_1 : (\bar{s}_1 : \Phi_{i,1}) \to P\ \bar{v}_{i,1}\ (x_1\ \bar{s}_1)) \to \ldots \to \\ (h_{n_i} : (\bar{s}_{n_i} : \Phi_{i,n_i}) \to P\ \bar{v}_{i,n_i}\ (x_{n_i}\ \bar{s}_{n_i})) \to \\ P\ \bar{u}_i\ (\texttt{c}_i\ \bar{t}\ x_1\ \ldots\ x_{n_i})\end{aligned} \tag{24}$$

**Elimination operators.** Datatype eliminators are an instance of the more general concept of an *elimination operator*. For any telescope $\Xi$, we define a $\Xi$-elimination operator (McBride, 2002) to be any function with a type of the form

$$\begin{aligned}(P : \Xi \to \texttt{Set}_i) \to \\ (m_1 : \Delta_1 \to P\ \bar{s}_1) \ldots (m_n : \Delta_n \to P\ \bar{s}_n) \to \\ (\bar{t} : \Xi) \to P\ \bar{t}\end{aligned} \tag{25}$$

We call $\Xi$ the *target*, $P$ the *motive*, and $m_1, \ldots, m_n$ the *methods* of the elimination operator. The reader may think of a $\Xi$-elimination operator as a way to transform a

[7] In the original definition of indexed families by Dybjer (1991), parameters are required to occur uniformly everywhere in the definition of the datatype, whilst indices can vary from constructor to constructor. Agda is less restrictive and also allows parameters to occur non-uniformly in the types of recursive constructor arguments, but not in their return types.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

problem into a set of subproblems. In the type shown above, the original problem is to construct a result of type $P\ \bar{t}$ when given arbitrary values $\bar{t}$ in the telescope $\Xi$. This original problem is transformed into $n$ sub-problems given by each of the methods: The $i$th subproblem is to construct a result of type $P\ \bar{s}_i$ when given arbitrary values of type $\Delta_i$. The elimination operator's type can be read as a function that transforms solutions for the sub-problems into a solution for the original problem.

**Basic analysis.** Note that a $\Xi$-elimination operator returns something of type $(\bar{u} : \Xi) \to P\ \bar{u}$ when given a motive $P : \Xi \to \mathtt{Set}_j$. However, we often need a return type where the arguments $\bar{u}$ are more specialized, for example, to construct a function of type $(k : \mathbb{N})(y : k \leqslant \mathtt{zero}) \to \mathtt{zero} \equiv k$. McBride (2002) solves this problem by adding the constraints on the indices as additional arguments to the motive $P$, and filling in $\overline{\mathtt{refl}}$ as soon as the constraints are satisfied. This technique is called *basic analysis*. For example, let $\mathtt{case}_{\leqslant}$ be the standard eliminator for $m \leqslant n$ with its recursive arguments dropped:

$$\begin{aligned}\mathtt{case}_{\leqslant} : (P &: (m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to \mathtt{Set}_i) \to \\ &(m_{\mathtt{lz}} : (m : \mathbb{N}) \to P\ \mathtt{zero}\ m\ (\mathtt{lz}\ m)) \to \\ &(m_{\mathtt{ls}} : (m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to P\ (\mathtt{suc}\ m)\ (\mathtt{suc}\ n)\ (\mathtt{ls}\ m\ n\ x)) \to \\ &(m : \mathbb{N})(n : \mathbb{N})(x : m \leqslant n) \to P\ m\ n\ x\end{aligned} \tag{26}$$

Then, the basic $\mathtt{case}_{\leqslant}$-analysis of $\mathtt{zero} \equiv k$ at $k; \mathtt{zero}; y$ has type

$$\begin{aligned}&(m_{\mathtt{lz}} : (m : \mathbb{N})(k : \mathbb{N})(y : k \leqslant \mathtt{zero}) \to \\ &\quad (\mathtt{zero}; m; \mathtt{lz}\ m) \equiv (k; \mathtt{zero}; y) \to \mathtt{zero} \equiv k) \to \\ &(m_{\mathtt{ls}} : (m\ n : \mathbb{N})(x : m \leqslant n)(k : \mathbb{N})(y : k \leqslant \mathtt{zero}) \to \\ &\quad (\mathtt{suc}\ m; \mathtt{suc}\ n; \mathtt{ls}\ m\ n\ x) \equiv (k; \mathtt{zero}; y) \to \mathtt{zero} \equiv k) \to \\ &(k : \mathbb{N})(y : k \leqslant \mathtt{zero}) \to \mathtt{zero} \equiv k\end{aligned} \tag{27}$$

Note that applying $\mathtt{case}_{\leqslant}$ directly to $y : k \leqslant \mathtt{zero}$ would lead to loss of the information that the second index of $y$ is $\mathtt{zero}$, thus leaving us unable to provide $m_{\mathtt{lz}}$ and $m_{\mathtt{ls}}$.

In general, let $\mathtt{elim}$ be any $\Xi$-elimination operator, and suppose we want to construct a function of type $\Delta \to \Phi$ by applying this eliminator to $\bar{t}$ where $\Delta \vdash \bar{t} : \Xi$. Then, we apply $\mathtt{elim}$ to the motive $\lambda(\bar{s} : \Xi).\ \Delta \to \bar{s} \equiv \bar{t} \to \Phi$. Filling in $\bar{t}$ for $\bar{s}$ and $\overline{\mathtt{refl}}$ for the proof of $\bar{s} \equiv \bar{t}$ gives us the *basic* $\mathtt{elim}$*-analysis of* $\Phi$ *at* $\bar{t}$:

$$\lambda m_1; \ldots; m_n; \bar{x}.\mathtt{elim}\ (\lambda \bar{s}.\ \Delta \to \bar{s} \equiv \bar{t} \to \Phi)\ m_1\ \ldots\ m_n\ \bar{t}\ \bar{x}\ \overline{\mathtt{refl}} \tag{28}$$

which is of type

$$(m_1 : \Delta_1 \Delta \to \bar{s}_1 \equiv \bar{t} \to \Phi) \ldots (m_n : \Delta_n \Delta \to \bar{s}_n \equiv \bar{t} \to \Phi) \to \Delta \to \Phi \tag{29}$$

Basic analysis will be used throughout the proof: once with $\mathtt{rec}_{\mathtt{D}}$ for structural recursion, and once with $\mathtt{case}_{\mathtt{D}}$ for each case split.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

## 5 A few homogeneous constructions on constructors

McBride *et al.* (2006) developed tools for working with inductive families of data types: case analysis, recursion, no confusion (subsuming both injectivity and disjointness of constructors), and acyclicity. In this section, we present these rules adapted to work with homogeneous instead of heterogeneous equality. Since the general form of these rules can be rather complex, we start by constructing them on an example datatype. Following the pedagogy of McBride *et al.* (2006), we take binary trees as our example data type (5.1). Once we have tackled this example, we will be equipped to handle the general case (5.2).

### *5.1 Example: binary trees*

As a first example, we consider the type `Tree` of binary trees, consisting of two constructors $\texttt{leaf} : \texttt{Tree}$ and $\texttt{node} : \texttt{Tree} \to \texttt{Tree} \to \texttt{Tree}$. The eliminator for `Tree` is

$$\begin{aligned}\texttt{elim}_{\texttt{Tree}} : (P : \texttt{Tree} \to \texttt{Set}_i) \to P\ \texttt{leaf} \to \\ ((l\ r : \texttt{Tree}) \to P\ l \to P\ r \to P\ (\texttt{node}\ l\ r)) \to (x : \texttt{Tree}) \to P\ x\end{aligned} \tag{30}$$

**Case Analysis.** Case analysis allows us to distinguish between the constructors of a datatype. In effect, it's just a weaker version of the standard eliminator, with the inductive hypotheses for the recursive arguments dropped. Here, it is for the `Tree` type:

$$\begin{aligned}\texttt{case}_{\texttt{Tree}} &: (P : \texttt{Tree} \to \texttt{Set}_i) \to P\ \texttt{leaf} \to \\ &\quad ((l\ r : \texttt{Tree}) \to P\ (\texttt{node}\ l\ r)) \to (x : \texttt{Tree}) \to P\ x \\ \texttt{case}_{\texttt{Tree}}\ &\ P\ m_{\texttt{leaf}}\ m_{\texttt{node}}\ t = \texttt{elim}_{\texttt{Tree}}\ P\ m_{\texttt{leaf}}\ (\lambda l\ r\ \_\ \_.\ m_{\texttt{node}}\ l\ r)\ x\end{aligned} \tag{31}$$

**Recursion.** Recursion allows us to prove things about trees by complete induction. It resembles the standard eliminator, but the inductive step allows us to assume $P\ t'$ for *all* subtrees of $t$, not just the direct ones. To construct the recursion principle, we first define a type $\texttt{Below}_{\texttt{Tree}}\ P\ x$, expressing that the property $P : \texttt{Tree} \to \texttt{Set}$ holds for any subtree of $x : \texttt{Tree}$. In other words, we have

$$\begin{aligned}\texttt{Below}_{\texttt{Tree}}\ P\ \texttt{leaf} &= \top \\ \texttt{Below}_{\texttt{Tree}}\ P\ (\texttt{node}\ l\ r) &= (\texttt{Below}_{\texttt{Tree}}\ P\ l \times P\ l) \times (\texttt{Below}_{\texttt{Tree}}\ P\ r \times P\ r)\end{aligned} \tag{32}$$

Second, the function $\texttt{below}_{\texttt{Tree}}$ encodes proof by complete induction on trees: if we can give a step function $s$ that proves $\texttt{Below}_{\texttt{Tree}}\ P\ t$ implies $P\ t$ for any tree $t$, then $\texttt{below}_{\texttt{Tree}}\ P\ s$ is a proof that $\texttt{Below}_{\texttt{Tree}}\ P\ t$ holds for any $t$. Finally, the recursion operator $\texttt{rec}_{\texttt{Tree}}\ P\ s$ applies the step function $s$ one more time to conclude $P\ t$ for any tree $t$.

**No Confusion.** The principle of no confusion packages two properties of constructors: injectivity and disjointness. In the case of `Tree`, injectivity allows us to conclude

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

$l_1 \equiv l_2$ and $r_1 \equiv r_2$ from $\texttt{node}\ l_1\ r_1 \equiv \texttt{node}\ l_2\ r_2$, and disjointness allows us to refute any equation of the form $\texttt{leaf} \equiv \texttt{node}\ l\ r$. To package these two principles together, we first define a type $\texttt{NoConfusion}\ t_1\ t_2$ that computes a goal type for two trees:

$$\begin{array}{llll} \texttt{NoConfusion} & \texttt{leaf} & \texttt{leaf} & = \top \\ \texttt{NoConfusion} & \texttt{leaf} & (\texttt{node}\ l\ r) & = \bot \\ \texttt{NoConfusion} & (\texttt{node}\ l\ r) & \texttt{leaf} & = \bot \\ \texttt{NoConfusion} & (\texttt{node}\ l_1\ r_1) & (\texttt{node}\ l_2\ r_2) & = l_1 \equiv l_2 \times r_1 \equiv r_2 \end{array} \tag{33}$$

The function $\texttt{noConf}_{\texttt{Tree}}$ gives a proof of $\texttt{NoConfusion}_{\texttt{Tree}}\ s\ t$ for any two trees $s$ and $t$ that are equal:

$$\texttt{noConf}_{\texttt{Tree}} : (s\ t : \texttt{Tree}) \to s \equiv t \to \texttt{NoConfusion}_{\texttt{Tree}}\ s\ t \tag{34}$$

It is constructed using the J eliminator and $\texttt{case}_{\texttt{Tree}}$. We will also need an inverse $\texttt{noConf}^{-1}_{\texttt{Tree}}$ of $\texttt{noConf}_{\texttt{Tree}}$ in order to type the `injectivity` rule in Section 6:

$$\texttt{noConf}^{-1}_{\texttt{Tree}} : (s\ t : \texttt{Tree}) \to \texttt{NoConfusion}_{\texttt{Tree}}\ s\ t \to s \equiv t \tag{35}$$

**Acyclicity.** Acyclicity for trees means that no tree can ever be a subtree of itself, i.e. all trees are well-founded. As for recursion and no confusion, we first define a type that states this property, and then prove it. The type we define is $x \not< t$, expressing that $x$ is *not* a subtree of $t$. It is defined using $\texttt{Below}_{\texttt{Tree}} : x \not< t = \texttt{Below}_{\texttt{Tree}}\ (\lambda s.x \not\equiv s)\ t$. Next, $\texttt{noCycle}_{\texttt{Tree}}$ is a proof that $t \not< t$ for any tree $t$:

$$\begin{array}{lll} \texttt{noCycle}_{\texttt{Tree}} & \texttt{leaf} & = \texttt{tt} \\ \texttt{noCycle}_{\texttt{Tree}} & (\texttt{node}\ l\ r) & = \texttt{step}_{\texttt{left}}\ (\texttt{noCycle}_{\texttt{Tree}}\ l), \texttt{step}_{\texttt{right}}\ (\texttt{noCycle}_{\texttt{Tree}}\ r) \end{array} \tag{36}$$

Here, $\texttt{step}_{\texttt{left}} : x \not< t \to \texttt{node}\ x\ s \not\leqslant t$ and $\texttt{step}_{\texttt{right}} : x \not< t \to \texttt{node}\ s\ x \not\leqslant t$ are two helper functions that can be defined using $\texttt{elim}_{\texttt{Tree}}$ and $\texttt{noConf}_{\texttt{Tree}}$.

### *5.2 The general case*

For the rest of this section, let $\texttt{D} : \Xi \to \texttt{Set}_i$ be an inductive family.

**Case Analysis.** $\texttt{case}_{\texttt{D}}$ is a weakened version of the standard eliminator $\texttt{elim}_{\texttt{D}}$ that we get by dropping the inductive hypotheses of the methods. $\texttt{case}_{\texttt{D}}$ is given by dropping the inductive hypotheses from the eliminator, i.e. it is itself a $\bar{\texttt{D}}$-elimination operator with methods:

$$m_i : (\bar{t} : \Delta_i) \to (x_1 : \Phi_{i,1} \to D\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to D\ \bar{v}_{i,n_i}) \to P\ \bar{u}_i\ (\texttt{c}_\texttt{i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \tag{37}$$

for $i = 1, \ldots k$.

**Recursion.** First, for $x : \texttt{D}\ \bar{u}$, $\texttt{Below}_{\texttt{D}}\ P\ \bar{u}\ x$ is a tuple type that is inhabited whenever $P\ \bar{v}\ y$ holds for all $y : \texttt{D}\ \bar{v}$ which are structurally smaller than $x : \texttt{D}\ \bar{u}$. In order to define $\texttt{Below}_{\texttt{D}}\ P$, we apply the eliminator $\texttt{elim}_{\texttt{D}}$ to the motive $\Phi = \lambda\ \_.\ \texttt{Set}_i$. For the

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

method $m_i$ corresponding to the constructor $\mathtt{c_i}$, we give the following:

$$\begin{aligned} m_i = \lambda \bar{t}\,; x_1 ; \ldots ; x_{n_i} ; h_1 ; \ldots ; h_{n_i}.\\ (\Phi_{i,1} \to h_1\ \Phi_{i,1} \times P\ \bar{v}_{i,1}\ (x_1\ \Phi_{i,1})) \times \\ \ldots \times (\Phi_{i,n_i} \to h_{n_i}\ \Phi_{i,n_i} \times P\ \bar{v}_{i,n_i}\ (x_{n_i}\ \Phi_{i,n_i})) \end{aligned} \tag{38}$$

i.e. $\mathtt{Below_D}\ P\ x$ is a tuple asserting $P\ y$ for all $y$ structurally smaller than $x$. Second, the helper function $\mathtt{below_D}$ constructs this tuple:

$$\begin{aligned} \mathtt{below_D} : (P : (\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Set}_i) \to ((\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Below_D}\ P\ \bar{x} \to P\ \bar{x}) \to \\ (\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Below_D}\ P\ \bar{x} \end{aligned} \tag{39}$$

To define $\mathtt{below_D}\ P\ p$, we apply $\mathtt{elim_D}$ with the motive $\mathtt{Below_D}\ P$. We give the following for the method $m_i$:

$$\begin{aligned} m_i = \lambda \bar{t}\,; x_1 ; \ldots ; x_{n_i} ; h_1 ; \ldots ; h_{n_i}.\\ (\lambda \Phi_{i,1}.\ h_1\ \Phi_{i,1},\ p\ \bar{v}_{i,1}\ x_1\ (h_1\ \Phi_{i,1})), \\ \ldots,\ (\lambda \Phi_{i,n_i}.\ h_{n_i}\ \Phi_{i,n_i},\ p\ \bar{v}_{i,n_i}\ x_{n_i}\ (h_{n_i}\ \Phi_{i,n_i})) \end{aligned} \tag{40}$$

Finally,

$$\mathtt{rec_D} : (P : (\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Set}_i) \to ((\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Below_D}\ P\ \bar{x} \to P\ \bar{x}) \to (\bar{x} : \bar{\mathtt{D}}) \to P\ \bar{x} \tag{41}$$

is used for well-founded recursion over values of type $\mathtt{D}$. It is defined as $\mathtt{rec_D}\ P\ p\ \bar{\mathtt{D}} := p\ \bar{\mathtt{D}}\ (\mathtt{below_D}\ P\ p\ \bar{\mathtt{D}})$.

**No Confusion.** First, $\mathtt{NoConfusion_D} : \bar{\mathtt{D}} \to \bar{\mathtt{D}} \to \mathtt{Set}_d$ is a type such that

$$\begin{aligned} \mathtt{NoConfusion_D}\ (\bar{u}\,; c\ \bar{s})\ (\bar{v}\,; c\ \bar{t}) &= \bar{s} \equiv \bar{t} \\ \mathtt{NoConfusion_D}\ (\bar{u}\,; c\ \bar{s})\ (\bar{v}\,; c'\ \bar{t}) &= \bot \qquad (\text{when } c \neq c') \end{aligned} \tag{42}$$

Note that the diagonal case (where we have two times the same constructor) $\mathtt{NoConfusion_D}$ only requires $\bar{s} \equiv \bar{t}$. From this, it follows that $\bar{u} \equiv \bar{v}$ as well, since the indices are determined by the constructor arguments.

Second, we construct

$$\mathtt{noConf_D} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \bar{x} \equiv \bar{y} \to \mathtt{NoConfusion_D}\ \bar{x}\ \bar{y} \tag{43}$$

We also construct an inverse

$$\mathtt{noConf_D}^{-1} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \mathtt{NoConfusion_D}\ \bar{x}\ \bar{y} \to \bar{x} \equiv \bar{y} \tag{44}$$

and give a proof $\mathtt{isLeftInv_D}$ that $(\mathtt{noConf_D}^{-1}\ \bar{x}\ \bar{y}) \circ (\mathtt{noConf}\ \bar{x}\ \bar{y})$ is the identity on $\bar{x} \equiv \bar{y}$.[8] The need for this inverse will become clear when we construct the unification transitions in Section 6.

To define $\mathtt{NoConfusion_D}\ \bar{a}\ \bar{b}$, we apply $\mathtt{case_D}$ with the motive $\lambda\ \_.\ \mathtt{Set}_i$ on $\bar{a}$. For each method $m_i\ \bar{x}$, we apply $\mathtt{case_D}$ again with the same motive, but this time on $\bar{b}$. This gives us $k^2$ methods $m_{i,j}$ to fill in, one for each pair of constructors. On the

[8] We could also prove that $(\mathtt{noConf_D}\ \bar{x}\ \bar{y}) \circ (\mathtt{noConf_D}^{-1}\ \bar{x}\ \bar{y})$ is the identity on $\mathtt{NoConfusion_D}\ \bar{x}\ \bar{y}$, thus establishing that $\mathtt{noConf_D}\ \bar{x}\ \bar{y}$ is an equivalence. However, this is not needed for the present work.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

diagonal (where $i = j$), we define $m_{ii} = \lambda \bar{x}; \bar{x}'.\ \bar{x} \equiv \bar{x}'$, and if $i \neq j$ we simply give $m_{i,j} = \lambda \bar{x}; \bar{x}'.\ \bot$ (the empty type).

Next, we define $\mathtt{noConf_D}\ \bar{a}\ \bar{b}$. To do this, we apply telescopic substitution $\overline{\mathtt{subst}}$ with motive $\mathtt{NoConfusion_D}\ \bar{a}$. We need to give a function of type

$$(\bar{a} : \bar{\mathtt{D}}) \to \mathtt{NoConfusion_D}\ \bar{a}\ \bar{a} \tag{45}$$

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

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

$$m_{i,j} : \mathtt{NoConfusion_D}\ (\bar{u}_i; \mathtt{c_i}\ \bar{x})\ (\bar{u}'_j; \mathtt{c_j}\ \bar{x}') \to \bar{u}_i\ (\mathtt{c_i}\ \bar{x}) \equiv \bar{u}'_j\ (\mathtt{c_j}\ \bar{x}') \tag{46}$$

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 $\bar{x} \equiv \bar{x}'$. Applying $\overline{\mathtt{subst}}$ to this equality leaves us the goal $\bar{u}'_j; (\mathtt{c_j}\ \bar{x}') \equiv \bar{u}'_j; (\mathtt{c_j}\ \bar{x}')$, which we can fill in with $\overline{\mathtt{refl}}$. This particular combination of $\overline{\mathtt{subst}}$ and $\overline{\mathtt{refl}}$ could be seen as a telescopic version of congruence on the mapping $\bar{x} \mapsto \bar{u}_j; c_j\ \bar{x}$.

Finally, we prove that this is indeed a (left) inverse by constructing a function of type

$$(\bar{a}\ \bar{b} : \bar{\mathtt{D}})(\bar{e} : \bar{a} \equiv \bar{b}) \to \mathtt{noConf_D}^{-1}\ \bar{a}\ \bar{b}\ (\mathtt{noConf_D}\ \bar{a}\ \bar{b}\ \bar{e}) \equiv \bar{e} \tag{47}$$

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

$$(\bar{a} : \bar{\mathtt{D}}) \to \mathtt{noConf_D}^{-1}\ \bar{a}\ \bar{a}\ (\mathtt{noConf_D}\ \bar{a}\ \bar{a}\ \overline{\mathtt{refl}}) \equiv \overline{\mathtt{refl}} \tag{48}$$

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

**Acyclicity.** First, $\bar{x} \not< \bar{y}$ is defined as a tuple type stating that $\bar{x} : \bar{\mathtt{D}}$ is not structurally smaller than $\bar{y} : \bar{\mathtt{D}}$. Second, $\mathtt{noCycle_D} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \bar{x} \equiv \bar{y} \to \bar{x} \not< \bar{y}$ states that no term can be structurally smaller than itself.

The relation $\not<$ is defined using $\mathtt{Below_D}$: $\bar{a} \not< \bar{b} := \mathtt{Below_D}\ (\lambda \bar{b}'.\ \bar{a} \not\equiv \bar{b}')\ \bar{b}$. We also define $\bar{a} \not\leqslant \bar{b} := \bar{a} \not< \bar{b} \times \bar{a} \not\equiv \bar{b}$. If $x : \mathtt{D}\ \bar{u}$ and $y : \mathtt{D}\ \bar{v}$, then we often write $x \not< y$ and $x \not\leqslant y$ instead of $\bar{u}; x \not< \bar{v}; y$ and $\bar{u}; x \not\leqslant \bar{v}; y$ to avoid having to write too much clutter. Note that $x \not< \mathtt{c_i}\ \Delta_i\ x_1\ \ldots\ x_{n_i} = (\Phi_{i,1} \to x \not\leqslant x_1\ \Phi_{i,1}) \times \cdots \times (\Phi_{i,n_i} \to x \not\leqslant x_{n_i}\ \Phi_{i,n_i})$ by definition of $\mathtt{Below_D}$ and $\not\leqslant$. Now to construct $\mathtt{noCycle_D}$, we start by eliminating the equation $\bar{a} \equiv \bar{b}$ using $\bar{\mathtt{J}}$, which leaves us the goal $(\bar{a} : \bar{\mathtt{D}}) \to \bar{a} \not< \bar{a}$. Next, we apply $\mathtt{elim_D}$ with motive $\lambda \bar{a}.\ \bar{a} \not< \bar{a}$, 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

$$\begin{aligned}
&(\bar{t} : \Delta_i) \to (x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1})\ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\
&(h_1 : \Phi_{i,1} \to x_1\ \Phi_{i,1} \not< x_1\ \Phi_{i,1})\ \ldots\ (h_{n_i} : \Phi_{i,n_i} \to x_{n_i}\ \Phi_{i,n_i} \not< x_{n_i}\ \Phi_{i,n_i}) \to \\
&\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not< \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}
\end{aligned} \tag{49}$$

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

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

$$\begin{aligned}\mathtt{Step_{i,j}} &: \Delta_i \to (x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1}) \ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ &\qquad \Phi_{i,j} \to \bar{\mathtt{D}} \to \mathtt{Set}_d \\ \mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \Phi_{i,j}\ (\bar{u}; b) &= (x_j\ \Phi_{i,j}) \not< b \to (\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i}) \not\leqslant b\end{aligned} \tag{50}$$

Now, suppose that we can construct

$$\begin{aligned}\mathtt{step_{i,j}} \quad : \quad & (\bar{t} : \Delta_i) \to (x_1 : \Phi_{i,1} \to \mathtt{D}\ \bar{v}_{i,1}) \ \ldots\ (x_{n_i} : \Phi_{i,n_i} \to \mathtt{D}\ \bar{v}_{i,n_i}) \to \\ & \Phi_{i,j} \to (\bar{a} : \bar{\mathtt{D}}) \to \mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \Phi_{i,j}\ \bar{a}\end{aligned} \tag{51}$$

Then, we can solve the subgoal by filling in

$$\begin{aligned}&\lambda \bar{t}; \bar{x}; \bar{h}. \\ &(\lambda \Phi_{i,1}.\ \mathtt{step_{i,1}}\ \bar{t}\ \bar{x}\ \Phi_{i,1}\ \bar{v}_{i,1}\ (x_1\ \Phi_{i,1})\ (h_1\ \Phi_{i,1})), \\ &\ldots, \\ &(\lambda \Phi_{i,n_i}.\ \mathtt{step_{in_i}}\ \bar{t}\ \bar{x}\ \Phi_{i,n_i}\ \bar{v}_{i,n_i}\ (x_{n_i}\ \Phi_{i,n_i})\ (h_{n_i}\ \Phi_{i,n_i}))\end{aligned} \tag{52}$$

So we only need to construct $\mathtt{step_{i,j}}$. The construction of $\mathtt{step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \Phi_{i,j}$ (of type $(\bar{a} : \bar{\mathtt{D}}) \to \mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \Phi_{i,j}\ \bar{a}$) proceeds by applying $\mathtt{elim_D}$ with the motive $\mathtt{Step_{i,j}}\ \bar{t}\ x_1\ \ldots\ x_{n_i}\ \Phi_{i,j}$. The new subgoals are of the form

$$\begin{aligned}&(\bar{t}' : \Delta'_p)(x'_1 : \Phi'_{p1} \to \mathtt{D}\ \bar{v}'_{p1}) \ \ldots\ (x'_{n_p} : \Phi'_{pn_p} \to \mathtt{D}\ \bar{v}'_{pn_p}) \to \\ &\quad (h_1 : (\bar{s}'_1 : \Phi'_{p1}) \to \mathtt{Step_{i,j}}\ \bar{t}\ \bar{x}\ \Phi_{i,j}\ \bar{v}'_{p1}\ (x'_1\ \bar{s}'_1)) \ \ldots \\ &\quad (h_{n_p} : (\bar{s}'_{n_p} : \Phi'_{pn_p}) \to \mathtt{Step_{i,j}}\ \bar{t}\ \bar{x}\ \Phi_{i,j}\ \bar{v}'_{pn_p}\ (x'_{n_p}\ \bar{s}'_{n_p})) \to \\ &\quad \mathtt{Step_{i,j}}\ \bar{t}\ \bar{x}\ \Phi_{i,j}\ \bar{u}'_p\ (\mathtt{c_p}\ \bar{t}'\ \bar{x}')\end{aligned} \tag{53}$$

We solve them by giving

$$\lambda \bar{t}'; x'_1; \ldots; x'_{n_p}; h_1; \ldots; h_{n_p}; H.\ \alpha, \beta \tag{54}$$

where we still have to construct

$$\alpha : \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not< \mathtt{c_p}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p} \tag{55}$$

and

$$\beta : \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not\equiv \mathtt{c_p}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p} \tag{56}$$

For any $\bar{s} : \Phi_{i,j}$, we have $H : x_j\ \bar{s} \not< \mathtt{c_p}\ \Delta'_p\ x'_1\ \ldots\ x'_{n_p}$ or, by definition of $\not<$, $H = (H_1,\ldots,H_{n_p})$, where $H_q : (\bar{s}' : \Phi'_{pq}) \to x_j\ \bar{s} \not\leqslant x'_q\ \bar{s}'$. The construction of $\alpha$ reduces to the construction of components $\alpha_q : \Phi'_{pq} \to \mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \not\leqslant x'_q\ \Phi'_{pq}$. But these we can give as $\alpha_q = \lambda \bar{s}'.\ h_q\ (\pi_1\ (H_p\ \bar{s}'))$ (where $\pi_1$ is projection onto the first component). For constructing $\beta$, we assume $\mathtt{c_i}\ \bar{t}\ x_1\ \ldots\ x_{n_i} \equiv \mathtt{c_p}\ \bar{t}'\ x'_1\ \ldots\ x'_{n_p}$ and derive an element of $\bot$. By $\mathtt{noConf_D}$, it suffices to consider the case where $i = p$, $\Delta_i = \Delta'_i$, and $x_1;\ldots;x_{n_i} = x'_1,\ldots,x'_{n_i}$. But then we have $H_j\ \bar{s} : x_j\ \bar{s} \not\leqslant x_j\ \bar{s}$, hence $\pi_2\ (H_j\ \Phi_{i,j})\ \overline{\mathtt{refl}} : \bot$. This finishes the construction of $\mathtt{noCycle_D}$.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

$$\begin{aligned}
&\texttt{solution} : (\Phi : (x : A)(e : x_0 \equiv x) \to \texttt{Set}_i)(m : \Phi\ x_0\ \texttt{refl})(x : A)(e : x_0 \equiv x) \to \Phi\ x\ e\\
&\texttt{solution}\ \ \Phi\ m\ x\ e = \mathrm{J}\ \Phi\ m\ x\ e
\end{aligned}$$

$$\begin{aligned}
&\texttt{injectivity} : (\Phi : (\bar{e} : \bar{u}_s; \texttt{c}\ \bar{s} \equiv \bar{u}_t; \texttt{c}\ \bar{t}) \to \texttt{Set}_i)\\
&\qquad (m : (\bar{e} : \bar{s} \equiv \bar{t}) \to \Phi\ (\texttt{noConf}_\texttt{D}{}^{-1}\ (\bar{u}_s; \texttt{c}\ \bar{s})\ (\bar{u}_t; \texttt{c}\ \bar{t})\ \bar{e})) \to\\
&\qquad (\bar{e} : \bar{u}_s; \texttt{c}\ \bar{s} \equiv \bar{u}_t; \texttt{c}\ \bar{t}) \to \Phi\ \bar{e}\\
&\texttt{injectivity}\ \ \Phi\ m\ \bar{e} = (\texttt{isLeftInv}_\texttt{D}\ (\bar{u}_s; \texttt{c}\ \bar{s})\ (\bar{u}_t; \texttt{c}\ \bar{t})\ \bar{e})_*\\
&\qquad\qquad (m\ (\texttt{noConf}_\texttt{D}\ (\bar{u}_s; \texttt{c}\ \bar{s})\ (\bar{u}_t; \texttt{c}\ \bar{t})\ \bar{e}))
\end{aligned}$$

$$\begin{aligned}
&\texttt{conflict} : (\Phi : (\bar{e} : \bar{u}_s; \texttt{c}_1\ \bar{s} \equiv \bar{u}_t; \texttt{c}_2\ \bar{t}) \to \texttt{Set}_i)(\bar{e} : \bar{u}_s; \texttt{c}_1\ \bar{s} \equiv \bar{u}_t; \texttt{c}_2\ \bar{t}) \to \Phi\ \bar{e}\\
&\texttt{conflict}\ \ \Phi\ \bar{e} = \texttt{elim}_\bot\ (\lambda_.\ \Phi\ \bar{e})\ (\texttt{noConf}_\texttt{D}\ (\bar{u}_s; \texttt{c}_1\ \bar{s})\ (\bar{u}_t; \texttt{c}_2\ \bar{t})\ \bar{e})
\end{aligned}$$

$$\begin{aligned}
&\texttt{cycle} : (\Phi : (\bar{e} : \bar{u}; x \equiv \bar{v}; \texttt{c}\ \bar{s}[x]) \to \texttt{Set}_i) \to (\bar{e} : \bar{u}; x \equiv \bar{v}; \texttt{c}\ \bar{s}[x]) \to \Phi\ \bar{e}\\
&\texttt{cycle}\ \ \Phi\ \bar{e} = \texttt{elim}_\bot\ (\lambda_.\ \Phi\ e)\ (\pi\ (\texttt{noCycle}_\texttt{D}\ (\bar{u}; x)\ (\bar{v}; \texttt{c}\ \bar{s}[x])\ \bar{e})\ \overline{\texttt{refl}})\\
&\qquad (\text{where } \pi : \bar{u}; x \not< \bar{v}; \texttt{c}\ \bar{s}[x] \to \bar{u}; x \not\equiv \bar{u}; x)
\end{aligned}$$

Fig. 8. The unification transitions represented as type-theoretic terms. Compared to the transitions given by Goguen *et al.* (2006), these work with the homogeneous equality and $\Phi$ has an additional dependence on the equality proof. Whilst these unification transitions are the most general ones we can construct, they are *not* the ones that we use for case splitting in practice. Rather, `injectivity`, `conflict`, and `cycle` are replaced by their more specialized variants `injectivity'` (61), `conflict'` (62), and `cycle'` (63).

## 6 Unification rules

In order to translate a node of the case tree to the application of an eliminator, we need terms that give an account of the unification process inside of type theory itself. In order to do this, we use the "no confusion" and "no cycle" properties from the previous section. This results in very general unification transitions; however, they can be difficult to apply in practice. So we also give more specialized versions of the transitions, which we will use for the proof in the next section.

### *6.1 An internal representation of the unification rules*

The unification transitions are given in Figure 8. Compared to Goguen *et al.* (2006), working with homogeneous equality leads us very naturally to upgraded unification transitions which are dependent on the equality proof. For example, consider a telescope $\Xi = (a : A)(b : B\ a)$ and a $\Xi$-elimination operator `elim`. Basic `elim`-analysis requires us to construct methods of type $\Delta \to a; b \equiv a'; b' \to T$, or if we expand the definition of telescopic equality:

$$\Delta \to (e_a : a \equiv a') \to (e_a)_*\ b \equiv b' \to T \tag{57}$$

The motive for eliminating $a \equiv a'$ is $(e_a)_*\ b \equiv b' \to T$, which depends on the proof $e_a$. So the dependence of $\Phi$ on the equality proofs is caused by the need to use substitution in the definition of homogeneous telescopic equality. Intuitively, it is

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

not surprising that not assuming *uniqueness of identity proofs* principle leads us to consider identity proofs relevant!

Note the lack of a `deletion` transition in Figure 8. The non-dependent version of `deletion` given by Goguen *et al.* (2006) has type

$$(\Phi : \mathtt{Set}_i) \to (m : \Phi) \to (e : x_0 \equiv x_0) \to \Phi \tag{58}$$

which can be constructed without K but would be quite useless in our situation because $\Phi$ cannot depend on $e$. In contrast, a dependent `deletion` rule would look like

$$\mathtt{deletion} : (\Phi : (e : x_0 \equiv x_0) \to \mathtt{Set}_i)(m : \Phi\ \mathtt{refl})(e : x_0 \equiv x_0) \to \Phi\ e \tag{59}$$

which is exactly the K axiom. This is the reason for the first restriction on the unification algorithm in our criterion, namely that the **deletion** rule cannot be used.

### 6.2 *Adapting the unification rules for practical use*

Another point of interest in Figure 8 is the type of $\Phi$ in the `injectivity` function: It is indexed over the equality proof of the indices $\bar{u}_s$ and $\bar{u}_t$ as well as the equality proof of $\mathtt{c}\ \bar{s}$ and $\mathtt{c}\ \bar{t}$. Whilst this is the most general form of injectivity that we can construct, it is quite difficult to apply in practice. This is because we need equations on each index of the datatype in addition to the equation between the constructors. This is in contrast to the **injectivity** rule from Section 2.3.

To get a function corresponding exactly to the **injectivity** rule, we need a more specialized version of `injectivity`, where the indices $\bar{u}_s$ and $\bar{u}_t$ are already definitionally equal:

$$\begin{aligned}\mathtt{injectivity}_{\mathtt{bad}} : (\Phi : (e : \mathtt{c}\ \bar{s} \equiv \mathtt{c}\ \bar{t}) \to \mathtt{Set}_i)(m : (\bar{e} : \bar{s} \equiv \bar{t}) \to \Phi\ ???)) \to \\ (e : \mathtt{c}\ \bar{s} \equiv \mathtt{c}\ \bar{t}) \to \Phi\ e\end{aligned} \tag{60}$$

However, unlike `injectivity` such a function can *not* be constructed from $\mathtt{noConf}_{\mathtt{D}}$. This is because in order to fill in the question marks, we need a function $g : \bar{s} \equiv \bar{t} \to \mathtt{c}\ \bar{s} \equiv \mathtt{c}\ \bar{t}$ such that we can prove $g\ (\mathtt{noConf}_{\mathtt{D}}\ (\bar{u}_s ; \mathtt{c}\ \bar{s})\ (\bar{u}_t ; \mathtt{c}\ \bar{t})\ \overline{\mathtt{refl}}\ e) \equiv e$ for arbitrary $e$, but no such $g$ can be found. In fact, wrongly using this transition caused a bug in Agda's `--without-K` option, allowing one to prove a weaker version of the K axiom (Cockx, 2014).

What we *can* construct from $\mathtt{noConf}_{\mathtt{D}}$ is the following:

$$\begin{aligned}\mathtt{injectivity'} : (\Phi : (\bar{e} : \bar{u} ; \mathtt{c}\ \bar{s} \equiv \bar{u} ; \mathtt{c}\ \bar{t}) \to \mathtt{Set}_i) \to \\ (m : (\bar{e} : \bar{s} \equiv \bar{t}) \to \Phi\ (\mathtt{noConf}_{\mathtt{D}}^{-1}\ (\bar{u} ; \mathtt{c}\ \bar{s})\ (\bar{u} ; \mathtt{c}\ \bar{t})\ \bar{e})) \to \\ (e : \mathtt{c}\ \bar{s} \equiv \mathtt{c}\ \bar{t}) \to \Phi\ \overline{\mathtt{refl}}\ e\end{aligned} \tag{61}$$

This rule is simply a specialized version of the `injectivity` rule in Figure 8. However, there is still a problem with this rule. Suppose we want to use it to construct a function of type $(e : \mathtt{c}\ \bar{s} \equiv \mathtt{c}\ \bar{t}) \to \Phi'\ e$, where $\Phi' : \mathtt{c}\ \bar{s} \equiv \mathtt{c}\ \bar{t} \to \mathtt{Set}_i$, and we want to apply `injectivity'`. Then, we need to find $\Phi : \bar{u} ; \mathtt{c}\ \bar{s} \equiv \bar{u} ; \mathtt{c}\ \bar{t} \to \mathtt{Set}_i$ such that $\Phi\ \overline{\mathtt{refl}}\ e = \Phi'\ e$ for arbitrary $e : \mathtt{c}\ \bar{s} \equiv \mathtt{c}\ \bar{t}$. This is problematic because we cannot eliminate the equations $\bar{u} \equiv \bar{u}$ in general without using the K axiom. This is

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

the reason for the second restriction on the unification algorithm in our criterion, namely that the indices $\bar{u}$ should be *self-unifiable*, i.e. specialization by unification of $\bar{u}$ with itself should succeed positively (see below for the definition of specialization by unification). This condition guarantees that we can construct $\Phi$ from $\Phi'$ by applying the unification transitions used in the self-unification of $\bar{u}$.

At first sight, the **conflict** and **cycle** rule suffer from the same problem as the **injectivity** rule because their motive $\Phi$ depends on the proof of $\bar{u}_s \equiv \bar{u}_t$ as well. However, in these cases, the problem can be solved because both `conflict` and `cycle` factor through the empty type $\bot$. To illustrate this, suppose we want to construct a function of type $(e : \mathtt{c}_1\ \bar{s} \equiv \mathtt{c}_2\ \bar{t}) \to \Phi'\ e$. First, we apply `conflict` with $\Phi = \lambda\bar{e}.\ \bot$, giving us a function of type $(\bar{e} : \bar{u}; \mathtt{c}_1\ \bar{s} \equiv \bar{u}; \mathtt{c}_2\ \bar{t}) \to \bot$. Filling in $\overline{\mathtt{refl}}$ for the equations $\bar{u} \equiv \bar{u}$ gives us $(e : \mathtt{c}_1\ \bar{s} \equiv \mathtt{c}_2\ \bar{t}) \to \bot$. Now by $\bot$-elimination, we also get a function $(e : \mathtt{c}_1\ \bar{s} \equiv \mathtt{c}_2\ \bar{t}) \to \Phi'\ e$. This gives us the following rule:

$$\mathtt{conflict'} : (\Phi : (e : \mathtt{c}_1\ \bar{s} \equiv \mathtt{c}_2\ \bar{t}) \to \mathtt{Set}_i)(e : \mathtt{c}_1\ \bar{s} \equiv \mathtt{c}_2\ \bar{t}) \to \Phi\ e \tag{62}$$

Analogously we can construct a function

$$\mathtt{cycle'} : (\Phi : (e : x \equiv \mathtt{c}\ \bar{s}[x]) \to \mathtt{Set}_i)(e : x \equiv \mathtt{c}\ \bar{s}[x]) \to \Phi\ \bar{e} \tag{63}$$

In our proof, we will use the primed variants `injectivity'`, `conflict'`, and `cycle'`.

### 6.3 Specialization by unification

Given any type of the form $\Delta \to \bar{u} \equiv \bar{v} \to T$ (for example, the types of $m_1, \ldots, m_n$ in the basic $\mathtt{case}_{\mathbb{D}}$ analysis), we may seek to construct an inhabitant of this type, called a *specializer*, by exhaustively iterating the unification transitions as applicable. Note that the shape of the first equation in $\bar{u} \equiv \bar{v}$ uniquely determines which unification rule applies (if any), so the unification process is deterministic. In case of a positive success, a specializer is found together with a substitution $\sigma : \Delta' \to \Delta$, given some $m : \Delta' \to T\sigma$. In the case of a negative success, a specializer is found without any additional assumptions.

As an example, suppose we want to construct a function of type

$$f : (n : \mathbb{N}) \to (e_1 : n \equiv \mathtt{zero})(e_2 : e_1^*\ (\mathtt{lz}\ n) \equiv \mathtt{lz}\ \mathtt{zero}) \to T\ n\ e_1\ e_2 \tag{64}$$

i.e. we have $\Delta = (n : \mathbb{N})$, $\bar{u} = n; (\mathtt{lz}\ n)$ and $\bar{v} = \mathtt{zero}; (\mathtt{lz}\ \mathtt{zero})$ (of type $(m : \mathbb{N})(x : \mathtt{zero} \leqslant m)$). The first unification step is `solution`, solving the variable $n$ to `zero` and leaving the following goal:

$$(e_2 : \mathtt{lz}\ \mathtt{zero} \equiv \mathtt{lz}\ \mathtt{zero}) \to T\ \mathtt{zero}\ \mathtt{refl}\ e_2 \tag{65}$$

The next step is `injectivity'`, but as noted in Section 6.2, this step requires us to check first that the indices of `lz zero` are self-unifiable. In this case, we have `lz zero` : `zero` $\leqslant$ `zero`, so the indices are `zero`;`zero`, which can clearly be unified with themselves by applying `injectivity` twice (note that `injectivity` is the same as `injectivity'` in this case because $\mathbb{N}$ is not indexed). So by `injectivity` on $e_2$,

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

we can simplify the equation further to

$$(e_2' : \texttt{zero} \equiv \texttt{zero}) \to T\ \texttt{zero}\ \texttt{refl}\ (\texttt{noConf}_{\leqslant}{}^{-1}\ (\texttt{zero};\texttt{zero};(\texttt{lz}\ \texttt{zero}))\ (\texttt{zero};\texttt{zero};(\texttt{lz}\ \texttt{zero}))\ e_2') \tag{66}$$

Finally, we apply `injectivity'` to $e_2'$ to simplify the goal to its final form:

$$T\ \texttt{zero}\ \texttt{refl}\ \texttt{refl} \tag{67}$$

The result of the unification process is the substitution $\sigma = [n \mapsto \texttt{zero}]$ and the specializer $f$ of the required type, where the remaining goal is $m : T\ \texttt{zero}\ \texttt{refl}\ \texttt{refl}$:

$$\begin{aligned} &f : (n : \mathbb{N}) \to (e_1 : n \equiv \texttt{zero})(e_2 : e_1^*\ (\texttt{lz}\ n) \equiv \texttt{lz}\ \texttt{zero}) \to T\ n\ e_1\ e_2 \\ &f = \texttt{solution}\ \Phi_1\ (\texttt{injectivity'}\ \Phi_2\ (\texttt{injectivity}\ \Phi_3\ m)) \end{aligned} \tag{68}$$

where

$$\begin{aligned} \Phi_1 &= \lambda n.\ \lambda e_1.\ (e_2 : \texttt{lz}\ n \equiv \texttt{lz}\ \texttt{zero}) \to T\ n\ e_1\ e_2 \\ \Phi_2 &= \texttt{injectivity}\ \Phi_m\ (\texttt{injectivity}\ \Phi_n\ (\lambda e_2'.\ T\ \texttt{zero}\ \texttt{refl}\ e_2)) \\ \Phi_m &= \lambda e_m.\ (e_n : \texttt{zero} \equiv \texttt{zero})(e_2 : (e_m;e_n)_*\ (\texttt{lz}\ \texttt{zero}) \equiv \texttt{lz}\ \texttt{zero}) \to \texttt{Set} \\ \Phi_n &= \lambda e_n.\ (e_2 : (e_n)_*\ (\texttt{lz}\ \texttt{zero}) \equiv \texttt{lz}\ \texttt{zero}) \to \texttt{Set} \\ \Phi_3 &= \lambda e_2' \to T\ \texttt{zero}\ \texttt{refl}\ (\texttt{noConf}_{\texttt{D}}{}^{-1}\ (\texttt{zero};\texttt{zero};(\texttt{lz}\ \texttt{zero}))\ (\texttt{zero};\texttt{zero};(\texttt{lz}\ \texttt{zero}))\ e_2') \end{aligned} \tag{69}$$

**Evaluation behaviour of the specializer.** There is something more we can say about the evaluation behaviour of specializers when applied to $\overline{\texttt{refl}}$:

*Lemma 1*
If specialization by unification delivers a substitution $\sigma : \Delta' \to \Delta$ and a specializer $s$ satisfying

$$(m : \Delta' \to T\sigma) \vdash s : \Delta \to \bar{u} \equiv \bar{v} \to T \tag{70}$$

then we have $s\ (\sigma\ \bar{t})\ \overline{\texttt{refl}} \rightsquigarrow^*\ m\ \bar{t}$ for any $\bar{t} : \Delta'$.

*Proof*
Note that the specializer $s$ consist of a series of applications $s_1\ (s_2\ \ldots(s_n\ m)\ldots)$, where each $s_i : (\Delta_{i+1} \to \bar{u}_{i+1} \equiv \bar{v}_{i+1} \to T\sigma_{i+1}) \to (\Delta_i \to \bar{u}_i \equiv \bar{v}_i \to T\sigma_i)$ is either `solution` $\Phi_i$[9] or `injectivity'` $\Phi_i$, as the other rules cannot occur since they don't require a method $m$. Here, we have that $\sigma_i : \Delta_i \to \Delta$ such that $\Delta_1 = \Delta$, $\bar{u}_1 = \bar{u}$, $\bar{v}_1 = \bar{v}$, $\sigma_1 = id$, $\Delta_{n+1} = \Delta'$, $\bar{u}_{n+1} = \bar{v}_{n+1} = ()$, $\sigma_{n+1} = \sigma$, and each $\sigma_{i+1}$ can be written as $\sigma_i \circ \tau_i$ for $\tau_i : \Delta_{i+1} \to \Delta_i$ (i.e. $\sigma = \tau_1 \circ \tau_2 \circ \ldots \circ \tau_n$). Then, it is sufficient to prove that for each $s_i$, we have $s_i\ k\ (\tau_i\ \bar{t})\ \overline{\texttt{refl}} \rightsquigarrow k\ \bar{t}\ \overline{\texttt{refl}}$ for arbitrary $\bar{t} : \Delta_{i+1}$ and $k : \Delta_{i+1} \to \bar{u}_{i+1} \equiv \bar{v}_{i+1} \to T\sigma_{i+1}$.

- For $s_i = \texttt{solution}\ \Phi_i$, we have $\Delta_i = (x : A)\Delta_{i+i}$ and $\tau_i\ \bar{x} = s;\bar{x}$ for some $s : A$. So by the computation rule for applying J to `refl`, we have $s_i\ k\ (\tau_i\ \bar{t})\ \overline{\texttt{refl}} = \mathrm{J}\ \Phi\ k\ (\tau_i\ \bar{t})\ \overline{\texttt{refl}} \rightsquigarrow k\ \bar{t}\ \overline{\texttt{refl}}$.

[9] Possibly composed with a dependency-preserving permutation of the arguments.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

- For $s_i = \texttt{injectivity'}\ \Phi_i$, we have $\Delta_i = \Delta_{i+1}$ and $\tau_i = id$. As `injectivity'` is constructed from `noConf` and `isLeftInv`, and both these functions map $\overline{\texttt{refl}}$ to $\overline{\texttt{refl}}$, we have $s_i\ k\ (\tau_i\ \bar{t})\ \overline{\texttt{refl}} \rightsquigarrow k\ \bar{t}\ \overline{\texttt{refl}}$.

So we have

$$\begin{aligned} s\ (\sigma\ \bar{t})\ \overline{\texttt{refl}} &= s_1\ (s_2\ \ldots(s_n\ m)\ldots)\ (\tau_1\ (\tau_2\ \ldots(\tau_n\ \bar{t})\ldots))\ \overline{\texttt{refl}} \\ &\rightsquigarrow (s_2\ \ldots(s_n\ m)\ldots)\ (\tau_2\ \ldots(\tau_n\ \bar{t})\ldots)\ \overline{\texttt{refl}} \\ &\rightsquigarrow \ldots \rightsquigarrow m\ \bar{t} \end{aligned} \tag{71}$$

as we wanted to prove. □

## 7 Eliminating pattern matching without K

In this section, we will prove our main theorem (see Section 3.4), i.e. we will show that definitions by dependent pattern matching satisfying our criterion can be translated to type theory with universes and inductive families, without using the K axiom. So let $\texttt{f} : (\bar{t} : \Delta) \to T$ be a function given by a valid case tree. As a running example, let $\texttt{f} = \texttt{antisym}$ from Definition (5). For this example, we have $\Delta = (m\ n : \mathbb{N})(x : m \leqslant n)(y : n \leqslant m)$ and $T = m \equiv n$.

*Proof*

Without loss of generality, let `f` be structurally recursive on some $t_j : \texttt{D}\ \bar{v}$, the $j$th variable in $\Delta$, where `D` is a datatype. In our example, `antisym` is structurally recursive on all four arguments, so we arbitrarily choose to do structural recursion on $x : m \leqslant n$. The basic $\texttt{rec}_\texttt{D}$-analysis of $T$ at $\bar{v}; t_j$ is

$$\lambda m^s; \bar{t}.\ \texttt{rec}_\texttt{D}\ P\ m^s\ (\bar{v}; t_j)\ \bar{t}\ \overline{\texttt{refl}} \tag{72}$$

which has type

$$(m^s : (\bar{x} : \bar{\texttt{D}}) \to \texttt{Below}_\texttt{D}\ P\ \bar{x} \to P\ \bar{x}) \to (\bar{t} : \Delta) \to T \tag{73}$$

where $P = \lambda \bar{x}.\ (\bar{t} : \Delta) \to \bar{x} \equiv \bar{v}; t_j \to T$. In our example, we have $P = \lambda m'; n'; x'.\ \Delta \to (m'; n'; x') \equiv (m; n; x) \to m \equiv n$.

Suppose we have an $m : (\bar{t} : \Delta) \to \texttt{Below}_\texttt{D}\ P\ (\bar{v}; t_j) \to T$, then we construct $m^s : (\bar{x} : \bar{\texttt{D}}) \to \texttt{Below}_\texttt{D}\ P\ \bar{x} \to (\bar{t} : \Delta) \to \bar{x} \equiv \bar{v}; t_j \to T$ by applying the telescopic equality eliminator $\bar{\texttt{J}}$ on the equations $\bar{x} \equiv \bar{v}; t_j$. More precisely, $m^s$ is defined as

$$\lambda \bar{x}; H; \bar{t}; \bar{e}.\ \bar{\texttt{J}}\ (\lambda \bar{x}; \bar{e}.\ \texttt{Below}_\texttt{D}\ P\ \bar{x} \to T)\ (m\ \bar{t})\ (\texttt{sym}\ \bar{e})\ H \tag{74}$$

where $\texttt{sym} : \bar{x} \equiv \bar{y} \to \bar{y} \equiv \bar{x}$. By Section 6.3, for any $\bar{t} : \Delta$, we have

$$m^s\ (\bar{v}; t_j)\ H\ \bar{t}\ \overline{\texttt{refl}} \rightsquigarrow m\ \bar{t}\ H \tag{75}$$

We will define `f'` as

$$\lambda \bar{t}.\ \texttt{rec}_\texttt{D}\ P\ m^s\ (\bar{v}; t_j)\ \bar{t}\ \overline{\texttt{refl}} : (\bar{t} : \Delta) \to T \tag{76}$$

once we have constructed a suitable $m$. Note that $m$ may make "recursive calls" to `f'` on arguments structurally smaller than $t_j$ using its argument of type $\texttt{Below}_\texttt{D}\ P\ (\bar{v}; t_j)$.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

Also, note that

$$\begin{aligned}\texttt{f'}\ \bar{t} &\rightsquigarrow^* \texttt{rec}_\texttt{D}\ P\ m^s\ (\bar{v};t_j)\ \bar{t}\ \overline{\texttt{refl}} \\ &\rightsquigarrow^* m^s\ (\bar{v};t_j)\ (\texttt{below}_\texttt{D}\ P\ m^s\ (\bar{v};t_j))\ \bar{t}\ \overline{\texttt{refl}} \\ &\rightsquigarrow^* m\ \bar{t}\ (\texttt{below}_\texttt{D}\ P\ m^s\ (\bar{v};t_j))\end{aligned} \tag{77}$$

In order to construct $m$, we proceed by induction on the structure of f's case tree. So suppose that we have arrived at some node with label $\bar{p}$, where $\bar{p}$ has pattern variables from a telescope $\Theta$ and we wish to construct $m : \Theta \to \texttt{Below}_\texttt{D}\ P\ (\bar{v};t_j)\tau \to T\tau$, where $\tau = [\Delta \mapsto \lceil \bar{p} \rceil]$[10]. Note that we have $\Theta = \Delta$ at the root node. There are three cases:

**Internal node.** In this case, the telescope is split on some variable $y$, where $\Theta = \Theta_1(y : \texttt{D'}\ \bar{v}_y)\Theta_2$ and D' is an inductive family. The basic $\texttt{case}_\texttt{D'}$ analysis of $\texttt{Below}_\texttt{D}\ P\ (\bar{v};t_j)\tau \to T\tau$ at $\bar{v}_y;y$ has type

$$\begin{aligned}&\ldots \to \\ &(m_\texttt{c} : (\bar{s} : \Delta_\texttt{c}) \to \Theta \to \bar{u}_s;\texttt{c}\ \bar{s} \equiv \bar{v}_y;y \to \\ &\quad \texttt{Below}_\texttt{D}\ P\ (\bar{v};t_j)\tau \to T\tau) \to \\ &\ldots \to \\ &\Theta \to \texttt{Below}_\texttt{D}\ P\ (\bar{v};t_j)\tau \to T\tau\end{aligned} \tag{78}$$

where there is one method $m_\texttt{c}$ for each constructor c of D'. In our example, the first case split is on $x : m \leqslant n$, and the basic $\texttt{case}_\leqslant$ analysis has type

$$\begin{aligned}&(m_\texttt{lz} : (k\ m\ n : \mathbb{N})(x : m \leqslant n)(y : n \leqslant m) \to (\texttt{zero};k;\texttt{lz}\ k) \equiv (m;n;x) \to \\ &\qquad \texttt{Below}\ P\ m\ n\ x \to m \equiv n) \\ &(m_\texttt{ls} : (k\ l : \mathbb{N})(u : k \leqslant l)(m\ n : \mathbb{N})(x : m \leqslant n)(y : n \leqslant m) \to \\ &\qquad (\texttt{suc}\ k;\texttt{suc}\ l;\texttt{ls}\ k\ l\ u) \equiv (m;n;x) \to \texttt{Below}\ P\ m\ n\ x \to m \equiv n) \\ &(m\ n : \mathbb{N})(x : m \leqslant n)(y : n \leqslant m) \to \texttt{Below}\ P\ m\ n\ x \to m \equiv n\end{aligned} \tag{79}$$

To construct the methods $m_\texttt{c}$, we apply specialization by unification on the equations $\bar{u}_s;\texttt{c}\ \bar{s} \equiv \bar{v}_y;y$, which we know will succeed by the definition of a valid case tree in Section 2.3. For the method $m_\texttt{lz}$ above, the first step is to apply solution to the equation $\texttt{zero} \equiv m$, simplifying the goal type to

$$\begin{aligned}m'_\texttt{lz} : (k\ n : \mathbb{N})(x : \texttt{zero} \leqslant n)(y : n \leqslant \texttt{zero}) \to (k;\texttt{lz}\ k) \equiv (n;x) \to \\ \texttt{Below}\ P\ \texttt{zero}\ n\ x \to \texttt{zero} \equiv n\end{aligned} \tag{80}$$

As another example, later on conflict is applied to the equation $\texttt{suc}\ l \equiv \texttt{zero}$ to construct a function

$$\begin{aligned}m_{\texttt{lz};\texttt{ls}} : (k\ l : \mathbb{N})(u : k \leqslant l)(y : \texttt{suc}\ k \leqslant \texttt{zero}) \to (\texttt{suc}\ l;\texttt{ls}\ k\ l\ u) \equiv (\texttt{zero};y) \to \\ \texttt{Below}_\leqslant\ P\ \texttt{zero}\ (\texttt{suc}\ k)\ (\texttt{lz}\ (\texttt{suc}\ k)) \to \texttt{zero} \equiv \texttt{suc}\ k\end{aligned} \tag{81}$$

[10] We define the operation $\lceil \bar{p} \rceil$ as taking a pattern $\bar{p}$ back to its underlying term, i.e. for a variable $x$, we have $\lceil x \rceil = x$, for a constructor c, we have $\lceil c\ p_1\ \ldots\ p_n \rceil = c\ \lceil p_1 \rceil\ \ldots \lceil p_n \rceil$ and for an inaccessible pattern $\lfloor t \rfloor$, we have $\lceil \lfloor t \rfloor \rceil = t$.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

For each $\mathtt{c}$ with positive success, we have to deliver a

$$m'_{\mathtt{c}} : \Theta' \to \mathtt{Below_D}\ P\ (\bar{v}; t_j)\tau\sigma \to T\tau\sigma \tag{82}$$

where $\sigma : \Theta' \to \Delta_{\mathtt{c}}\Theta$ is the substitution found by unification. But the inductive hypothesis for the subtree corresponding to the constructor $\mathtt{c}$ gives us exactly such a function. For $m_{\mathtt{lz}}$, the goal type becomes

$$m'''_{\mathtt{lz}} : (k : \mathbb{N})(y : k \leqslant \mathtt{zero}) \to \mathtt{Below}\ P\ \mathtt{zero}\ k\ (\mathtt{lz}\ k) \to \mathtt{zero} \equiv k \tag{83}$$

after applying `solution` two more times, at which point we proceed with another case split on $y$.

To determine the evaluation behaviour of $m$ when applied to a constructor, suppose we have some $\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2 : \Theta_1(y : \mathtt{D'}\ \bar{v}_y)\Theta_2$. Note that this means we must have $\bar{u}_s[\Delta_c \mapsto \bar{s}] = \bar{v}_y[\Theta_1 \mapsto \bar{t}_1]$, hence the equations $\bar{u}_s; \mathtt{c}\ \bar{s} \equiv \bar{v}_y; y$ are satisfied under the substitution mapping $\Delta_c$ to $\bar{s}$ and $\Theta$ to $\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2$. But $\sigma$ is the most general unifier of these equations, so there must exist some $\bar{t}' : \Theta'$ such that $\sigma\ \bar{t}' = \bar{s}; \bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2$. By Lemma 1, it follows that

$$\begin{aligned} m\ (\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2) &\rightsquigarrow^* m_{\mathtt{c}}\ \bar{s}\ (\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2)\ \overline{\mathtt{refl}} \\ &= m_{\mathtt{c}}\ (\sigma\ \bar{t}') \\ &\rightsquigarrow^* m'_{\mathtt{c}}\ \bar{t}' \end{aligned} \tag{84}$$

Moreover, by construction of the unifier, we have that $\bar{t}'$ consists of exactly those terms in $\bar{s}; \bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2$ that haven't been instantiated by unification.

**Empty node.** We follow the same construction as in the previous case, noting that all unifications will succeed negatively, hence no methods $m_{\mathtt{c}}$ are needed. Absurd clauses have no right-hand side, so they describe no reduction behaviour.

**Leaf node.** At each leaf node, we have the right-hand side $\Delta_i \vdash e_i : T\tau$. We wish to instantiate $m_i = \lambda\bar{s}; H.\ e_i$, but $e_i$ may still contain recursive calls to $\mathtt{f}$. In our example, the goal type for the second leaf node is

$$\begin{aligned} m_{\mathtt{ls}} : (k\ l : \mathbb{N})(u : k \leqslant l)(v : l \leqslant k) &\to \mathtt{Below}_{\leqslant}\ P\ (\mathtt{suc}\ k)\ (\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ u) \\ &\to \mathtt{suc}\ k \equiv \mathtt{suc}\ l \end{aligned} \tag{85}$$

and the right-hand side is $\mathtt{cong}\ \mathtt{suc}\ (\mathtt{antisym}\ k\ l\ u\ v)$. We first have to replace these recursive calls by appropriate calls to $H : \mathtt{Below_D}\ P\ (\bar{v}; t_j)\tau$. So consider a recursive call $\mathtt{f}\ \bar{r}$ in $e_i$. Since $\mathtt{f}$ is structurally recursive, we have $r_j \prec \lceil p_{i,j} \rceil$, where $r_j : D\ \bar{w}$. By construction of $\mathtt{Below_D}$, we have a projection $\pi$ such that $\pi\ H : (\bar{t} : \Delta) \to \bar{w}; r_j \equiv \bar{v}; t_j \to T$. Hence, we can define $e'_i$ by replacing $\mathtt{f}\ \bar{r}$ by $\pi\ H\ \bar{r}\ \overline{\mathtt{refl}} : T[\Delta \mapsto \bar{r}]$ in $e_i$, and take $m_i = \lambda\bar{s}; H.\ e'_i$. For `antisym`, we have

$$\pi_1\ H : (m\ n : \mathbb{N})(x : m \leqslant n)(y : n \leqslant m) \to (k; l; u) \equiv (m; n; x) \to m \equiv n \tag{86}$$

so we replace the recursive call $\mathtt{antisym}\ k\ l\ u\ v$ by $\pi_1\ H\ k\ l\ u\ v\ \overline{\mathtt{refl}}$. When we fill in $H = \mathtt{below_D}\ P\ m^s\ (\bar{v}; t_j)$, we get

$$\begin{aligned} &\pi\ (\mathtt{below_D}\ P\ m^s\ (\bar{v}; t_j))\ \bar{r}\ \overline{\mathtt{refl}} \\ &\quad\rightsquigarrow^* m^s\ (\bar{w}; r_j)\ (\mathtt{below_D}\ P\ m^s\ (\bar{w}; r_j))\ \bar{r}\ \overline{\mathtt{refl}} \\ &\quad\rightsquigarrow^* m\ \bar{r}\ (\mathtt{below_D}\ P\ m^s\ (\bar{w}; r_j)) = \mathtt{f'}\ \bar{r} \end{aligned} \tag{87}$$

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

By induction, we now have the required $m : (\bar{t} : \Delta) \to \texttt{Below}_\texttt{D}\ P\ \bar{v}\ x \to T$, thus finishing the construction of `f'`.

For each clause

$$\texttt{f}\ \bar{p}_i = e_i \tag{88}$$

with pattern variables $\bar{s} : \Delta_i$ at a leaf node of `f`'s case tree, we have

$$\begin{aligned}
\texttt{f'}\ \lceil \bar{p}_i \rceil &\rightsquigarrow^* m\ \lceil \bar{p}_i \rceil\ (\texttt{below}_\texttt{D}\ P\ m^s\ \bar{u}\ \lceil p_{i,j} \rceil) \\
&\rightsquigarrow^* m_\texttt{c}\ \ldots \text{ (working our way down the case tree)} \\
&\rightsquigarrow^* m_i\ \bar{s}\ (\texttt{below}_\texttt{D}\ P\ m^s\ \bar{u}\ \lceil p_{i,j} \rceil) \\
&\rightsquigarrow^* e_i'[H \mapsto \texttt{below}_\texttt{D}\ P\ m^s\ \bar{u}\ \lceil p_{i,j} \rceil] \\
&\rightsquigarrow^* \{e_i\}_\texttt{f}^\texttt{f'}
\end{aligned} \tag{89}$$

Hence, we can conclude that whenever $\texttt{f}\ \bar{t} \rightsquigarrow u$, we also have $\texttt{f'}\ \bar{t} \rightsquigarrow^* \{u\}_\texttt{f}^\texttt{f'}$, as we wanted to prove. □

## 8 Making pattern matching without K less restrictive

In Section 3.5, we remarked that our criterion was more general than the syntactic one. However, it still has some problems of its own. Suppose for example, we are working with the inequality $\leqslant$ indexed over finite sets `Fin` $n$, and we try to unify two successors in the same finite set. The problem $\texttt{fs}\ n\ x = \texttt{fs}\ n\ y$ requires solving $n = n$, but then we get stuck because we cannot use **deletion**. It can be proven that assuming K in general is not really needed for this example, so the criterion is still overly conservative. We now discuss a possible solution to handle cases like this one.

Looking back at the construction of the unification transitions in Section 6, we disallowed using **deletion** on an equation $x = x$ because *in general* this requires assuming K. However, for certain types of $x$, K can actually be proven without assuming it as an axiom. These types are called *(homotopy) sets* in HoTT. For example, $\mathbb{N}$ is a set (see Figure 9 for a proof of this fact), so it would be fine to use **deletion** on an equation $n = n$ when $n : \mathbb{N}$. This would already solve the problem described above.

The question then remains how to detect which types satisfy K and which do not. One possible solution is to require the user to prove K manually for a particular type, and then use this proof during unification by means of Agda's instance arguments (Devriese and Piessens, 2011). However, there is a problem with this approach: Agda does not keep any evidence of the unification performed whilst checking a definition by pattern matching. So if a proof of K is used that contains free variables whilst checking a function, the fact that the proof of K contains free variables is not reflected in the definition of that function. This could cause major problems with typechecking later on. Additionally, the proof of K used by unification also needs to satisfy a certain definitional behaviour: $\texttt{K}\ x\ P\ p\ \texttt{refl}$ has to evaluate to $p$. Otherwise, the definitional behaviour of the desugared function will disagree with the original definition.

Another approach that requires less user input and less checks is to try to automatically detect which types satisfy K. There is no general way to do this, but

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

$$\mathrm{K}_{\mathbb{N}} : (n : \mathbb{N})(P : n \equiv n \to \mathtt{Set}) \to P\ \mathtt{refl} \to (e : n \equiv n) \to P\ e$$
$$\mathrm{K}_{\mathbb{N}}\ \mathtt{zero}\ P\ p\ \mathtt{refl} = p$$
$$\mathrm{K}_{\mathbb{N}}\ (\mathtt{suc}\ n)\ P\ p\ e = \mathtt{subst}\ P\ (\mathtt{add\text{-}drop}\ e)\ (\mathrm{K}_{\mathbb{N}}\ n\ (P \circ \mathtt{add})\ p\ (\mathtt{drop}\ e))$$

**where**

$$\begin{array}{lcl} \mathtt{add} & : & n \equiv n \to \mathtt{suc}\ n \equiv \mathtt{suc}\ n \\ \mathtt{add} & = & \mathtt{noConf}_{\mathbb{N}}^{-1}\ (\mathtt{suc}\ n)\ (\mathtt{suc}\ n) \end{array}$$

$$\begin{array}{lcl} \mathtt{drop} & : & \mathtt{suc}\ n \equiv \mathtt{suc}\ n \to n \equiv n \\ \mathtt{drop} & = & \mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ n)\ (\mathtt{suc}\ n) \end{array}$$

$$\begin{array}{lcl} \mathtt{add\text{-}drop} & : & (e : \mathtt{suc}\ n \equiv \mathtt{suc}\ n) \to \mathtt{add}\ (\mathtt{drop}\ e) \equiv e \\ \mathtt{add\text{-}drop} & = & \mathtt{isLeftInv}_{\mathbb{N}}\ (\mathtt{suc}\ n)\ (\mathtt{suc}\ n) \end{array}$$

Fig. 9. A proof that the type $\mathbb{N}$ of natural numbers satisfies K, using dependent pattern matching with our criterion. The match on `refl` in the first clause passes our criterion because the unification problem is $\mathtt{zero} = \mathtt{zero}$, which can be solved by **injectivity**. The recursive call to $\mathrm{K}_{\mathbb{N}}$ in the second clause is permitted because the first argument decreases from $\mathtt{suc}\ n$ to $n$. We use the functions `noConf`, $\mathtt{noConf}^{-1}$, and `isLeftInv` constructed from eliminators in Section 5, but we could define these functions using pattern matching as well.

we could at least try to detect easy cases like $\mathbb{N}$. For example, let `D` be a simple (non-indexed) datatype such that each constructor is of the form $\mathtt{c} : \mathtt{D} \to \ldots \to \mathtt{D} \to \mathtt{D}$. Then, `D` has decidable equality, hence it is a set by Hedberg's theorem (Kraus *et al.*, 2013). More generally, we will prove the following theorem:

*Theorem 2*
Suppose that `D` is an inductive family with indices $\Xi$ and constructors $\mathtt{c_i} : \Delta_i \to \mathtt{D}\ \bar{v}_{i,1} \to \ldots \to \mathtt{D}\ \bar{v}_{i,n_i} \to \mathtt{D}\ \bar{u}_i$ (i.e. all recursive arguments are first-order). Suppose moreover that we already know all types in $\Xi$ and $\Delta_1, \ldots, \Delta_k$ to satisfy K. Then, $\mathtt{D}\ \bar{u}$ satisfies K as well for arbitrary $\bar{u}$.

This criterion can be used to reintroduce the **deletion** step of the unification algorithm on a more limited basis, namely to delete an equation $x = x$ only if the type of $x$ can be seen to be a set based on the criterion.

*Proof*
We will first construct a variant of K:

$$\mathrm{K}'_{\mathtt{D}} : (\bar{x} : \bar{\mathtt{D}})(\Phi : \bar{x} \equiv \bar{x} \to \mathtt{Set}_i) \to \Phi\ \overline{\mathtt{refl}} \to (\bar{e} : \bar{x} \equiv \bar{x}) \to \Phi\ \bar{e} \tag{90}$$

By $\mathtt{rec_D}$, during the construction of $\mathrm{K}'_{\mathtt{D}}$, we can assume to have a proof of $(\Phi : \bar{y} \equiv \bar{y} \to \mathtt{Set}_i) \to \Phi\ \overline{\mathtt{refl}} \to (\bar{e} : \bar{y} \equiv \bar{y}) \to \Phi\ \bar{e}$ for all $\bar{y} \prec \bar{x}$. We start by applying $\mathtt{elim_D}$ to $\bar{x}$, requiring us to provide for each constructor $\mathtt{c_i} : \Delta_i \to \mathtt{D}\ \bar{u}_i$ a method $m_i : (\bar{e} : \bar{u}_i; \mathtt{c_i}\ \bar{s}_i \equiv \bar{u}_i; \mathtt{c_i}\ \bar{s}_i) \to \Phi\ \bar{e}$, given $\Phi : (\bar{e} : \bar{u}_i; \mathtt{c_i}\ \bar{s}_i \equiv \bar{u}_i; \mathtt{c_i}\ \bar{s}_i) \to \mathtt{Set}_i$ and a proof $\phi : \Phi\ \overline{\mathtt{refl}}$. We construct $m_i$ by applying `injectivity` $\Phi$ (see Figure 8), reducing the goal to finding a $m'_i : (\bar{e} : \bar{s}_i \equiv \bar{s}_i) \to \Phi\ (\mathtt{noConf_D}^{-1}\ (\bar{u}_i; \mathtt{c_i}\ \bar{s}_i)\ (\bar{u}_i; \mathtt{c_i}\ \bar{s}_i)\ \bar{e})$. Now, note that the types of all $\bar{s}_i$ satisfy K: For the non-recursive arguments, this is true because they come from $\Delta_i$, and for the recursive arguments, this holds because of the inductive hypothesis. So by K for these types, we can assume $\bar{e}$ to be $\overline{\mathtt{refl}}$,

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

reducing the goal to $\Phi\ (\mathtt{noConf_D}^{-1}\ (\bar{u}_i; \mathtt{c_i}\ \bar{s}_i)\ (\bar{u}_i; \mathtt{c_i}\ \bar{s}_i)\ \overline{\mathtt{refl}}) \rightsquigarrow \Phi\ \overline{\mathtt{refl}}$. But this is exactly the type of $\phi$, completing the construction of $\mathrm{K_D}'$.

Finally, we construct $\mathrm{K_D} : (x : \mathtt{D}\ \bar{u})(\Phi : x \equiv x \to \mathtt{Set}_i) \to \Phi\ \mathtt{refl} \to (e : x \equiv x) \to \Phi\ e$ from $\mathrm{K'_D}$. Since the index types $\Xi$ of $\mathtt{D}$ all satisfy K, we can eliminate all but the last equation of $\bar{x} \equiv \bar{x}$ in the type of $\Phi$ to construct $\mathrm{K_D} : (x : \mathtt{D}\ \bar{u})(\Phi : x \equiv x \to \mathtt{Set}_i) \to \Phi\ \mathtt{refl} \to (e : x \equiv x) \to \Phi\ e$. More explicitly, $\mathrm{K_D}\ x\ \Phi\ \phi\ e = \mathrm{K'_D}\ (\bar{u}; x)\ \Phi'\ \phi\ (\overline{\mathtt{refl}}; e)$, where

$$\Phi' = \mathrm{K}_\Xi\ \bar{u}\ (\lambda \bar{e}.\ \overline{\mathtt{subst}}\ \mathtt{D}\ x\ \bar{u}\ \bar{e} \equiv x \to \mathtt{Set}_i)\ \Phi \tag{91}$$

and $\mathrm{K}_\Xi : (\bar{u} : \Xi)(\Phi : \bar{u} \equiv \bar{u} \to \mathtt{Set}_i) \to \Phi\ \overline{\mathtt{refl}} \to (\bar{e} : \bar{u} \equiv \bar{u}) \to \Phi\ \bar{e}$ can be constructed from the proofs of K for the individual types in $\Xi$. So this also concludes the construction of $\mathrm{K_D}$. □

Note that the term proving K constructed by this theorem does not satisfy the usual computational behaviour of the general K axiom 15 that $\mathrm{K}\ P\ p\ \mathtt{refl} \rightsquigarrow p$. The reason for this is that it is defined by induction on $\bar{x} : \bar{\mathtt{D}}$, so it is stuck if $\bar{x}$ is normal. As a consequence, if this proof of K is used for compiling a definition by pattern matching to eliminators, then Lemma 1 will fail to hold, and the result will not satisfy the reduction behaviour on open terms given in Theorem 1. However, any construction of K for a datatype that only uses the eliminator of that datatype will necessarily have to be recursive on the datatype, so there is no way to fix this problem without extending the theory with additional evaluation rules that go beyond the type theory presented in this paper.

## 9 Related work

Most implementations of dependent pattern matching in the style of Coquand (1992) do this by assuming the K axiom. Examples include Agda (when – without K is not enabled), Epigram (McBride and McKinna, 2004; McBride, 2005), Idris (Brady, 2013), the pattern matching construct for Coq described by Barras *et al.* (2009), and the Equations package for Coq described by Sozeau (2010).

Coq also support a more primitive notion of pattern matching via the `match` construct in Gallina (The Coq development team, 2012). The full version of this construct is

$$\begin{aligned}
&\mathtt{match}\ e\ \mathtt{as}\ x\ \mathtt{in}\ D\ \bar{u}\ \mathtt{return}\ P\ \mathtt{with}\\
&\quad \mid c_1\ \bar{y}_1 \Rightarrow e_1\\
&\quad \mid \ldots\\
&\quad \mid c_n\ \bar{y}_n \Rightarrow e_n\\
&\mathtt{end}
\end{aligned} \tag{92}$$

In the language of this paper, this corresponds to

$$\mathtt{case_D}\ (\lambda \bar{u}; x.\ P)\ (\lambda \bar{y}_1.\ e_1)\ \ldots\ (\lambda \bar{y}_n.\ e_n)\ e \tag{93}$$

Coq also allows skipping the parts labelled by `as`, `in`, and `return`, in which case it will attempt to construct the motive $P$ automatically.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

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

Recently, a new version of the Equations package for Coq has been developed that also supports pattern matching without assuming the K axiom (Mangin and Sozeau, 2015; Sozeau, 2015). Similarly to the work presented in this paper, it uses a generalization of homogeneous telescopic equality to achieve compilation of pattern matching definitions without K. In contrast to the implementation of our criterion in Agda, the Equations package also performs the actual translation of definitions by pattern matching to eliminators. It also allows the use of the **deletion** rule in some cases that are non-dependent or justified by user-provided instances of K.

The Lean theorem prover (de Moura *et al*., 2015) is a new dependently typed language that also supports dependently typed pattern matching by a translation to eliminators, similar to the Equations package. Lean can currently be used with two instantiations of its core theory: one based on the Calculus of Inductive Constructions which allows proving K, and a second one based on HoTT which doesn't. So using this second instantiation also allows one to use dependent pattern matching without relying on the K axiom. The authors cite the conference version of our current paper as an important source of inspiration for the implementation of the Lean system.

An unpublished first version of dependent pattern matching by McBride (1998) also used homogeneous equality with telescopic substitution and hence a proof-relevant unification algorithm. Similar to our present work, he observes that the innocent-looking **deletion** rule turns into the rather less innocent K. However, the published version of this work uses the heterogeneous equality, thus making it rely on K. This resulted in a significant simplification by avoiding dependency on equality proofs. In our current work, this extra complexity becomes a feature.

In his thesis, Boutillier (2014) describes an algorithm for compiling definitions by pattern matching to eliminators in Coq. The criterion he uses is very similar to the old criterion used by Agda: In order to perform a case distinction on a variable of an inductive family, the indices need to be constructors applied to distinct variables, and those variables must not occur in the parameters. To this, he adds a preprocessing step where indices are erased if they are not used in the return type or if they are determined by the type of the other indices. For the translation, he constructs a *diagonalizer* based on the skeleton of the indices, encoding the induction principle for a particular subset of the inductive family. Compared to our work, Boutillier doesn't give a closed criterion for when a definition by pattern matching is acceptable in a theory without K. Instead, he provides a desugaring of which the result still has to be checked by Coq. In our opinion, this is bad practice because it requires the user to be aware of about the desugaring in order to predict whether a definition will be accepted. In contrast, our criterion only requires the user to know the unification algorithm in order to predict its behaviour. The computational behaviour of the

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

desugaring also seems more like an afterthought in Boutillier's work, whilst it is an essential part of ours. Nevertheless, he does a better job than us of analysing whether an argument is actually used, either in the type of a later argument or in the return type. This gives a good heuristic for preprocessing pattern matching definitions in order to remove superfluous uses of K, so it could be used complementarily to our criterion.

## 10 Future work

**Automatic translation to eliminators.** One thing we noticed during the writing of this proof is how easily a small mistake can have grave impact on the soundness. For example, it was only after a long time that we realized just disabling **deletion** was not enough, but that the **injectivity** rule also subtly depends on K. To increase our confidence, we should make the typechecker of our languages perform the translation from pattern matching to a core calculus in practice. This is already done in Epigram (McBride and McKinna, 2004; McBride, 2005) and in the Equations package for Coq by Sozeau (2010). The latest version of the Equations package can also avoid using K (Sozeau, 2015). A very appealing idea to continue this line of work is to perform the compilation of dependent pattern matching to eliminators inside the type theory itself by means of *datatype-generic programming* as described by Dagand (2013), which would increase our confidence in the translation even further.

**Type class approach for user-provided instances of K.** As mentioned in Section 8, two problems prevent us from using the type class approach for user-provided instances of K in our current Agda implementation: These instances might contain free (meta-)variables, and they might not have the correct reduction behaviour. However, if Agda were to actually perform the desugaring of pattern matching to eliminators, it seems possible to soundly integrate such user-provided K instances into the desugared functions, even if they contain free variables or axioms. We expect that the resulting desugared functions would be type-safe, but their computational behaviour would depend on the computational behaviour of the user-provided K instances: If these do not reduce to `refl` when applied to `refl`, the function clauses would not hold definitionally. We think it could be interesting to investigate whether such a solution would be useful in practice.

**A global view on unification problems.** In the treatment of the unification rules in Section 6, we were forced to work with a specialized version of the **injectivity** rule rather than the fully general one. The main reason is that our current approach only deals with one equation at a time. In a new paper (Cockx *et al.*, to appear), we solve this limitation by taking a global view at unification problems and tracking how exactly the type of certain equations depends on others.

**Adaptation to cubical type theory.** The current version of our criterion (and the corresponding proof) are written for an Agda-like theory based on Unified Theory

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

of Dependent Types (Luo, 1994). In such a theory, principles such as functional extensionality or univalence can be postulated but they don't get any computational behaviour. On the other hand, a new and promising theory called *cubical* type theory (Bezem *et al.*, 2014; Cohen *et al.*, 2015) gives a constructive interpretation to the univalence axiom, and hence also functional extensionality. In the future, we would like to adapt the work in this paper to this setting, so our criterion would become usable in languages based on cubical type theory as well.

One obstacle for this adaptation is the fact that the representation of data types in our theory (and also that of Agda, Coq, Idris, ...) is computationally incompatible with functional extensionality. We will give an example to illustrate the problem.[11] Let $\texttt{Favourite} : (\mathbb{N} \to \mathbb{N}) \to \texttt{Set}$ be a data type with one constructor $\texttt{favourite} : \texttt{Favourite}\ (\lambda x.\ 0+x)$. We can give a proof $p$ of $(x : \mathbb{N}) \to 0+x \equiv x+0$, so we have $\texttt{ext}\ p : \lambda x.\ 0 + x \equiv \lambda x.\ x + 0$ and thence

$$\texttt{subst}\ \texttt{Favourite}\ (\texttt{ext}\ p)\ \texttt{favourite} : \texttt{Favourite}\ (\lambda x.\ x + 0) \tag{94}$$

However, there is no closed canonical form of type $\texttt{Favourite}\ (\lambda x.\ x+0)$ so this term doesn't reduce to a canonical form. This cannot be fixed by taking the constructor itself to be the canonical form (i.e. by letting $\texttt{favourite} : \texttt{Favourite}\ (\lambda x.\ x+0)$), as this would require the typechecker to check whether two functions are extensionally equal, which is undecidable in general.

This incompatibility could be solved by using a different internal representation of data types where each constructor carries explicit proofs of the constraints it imposes on the indices. For example, `favourite` would have the internal type $(e : f \equiv (\lambda x.\ 0 + x) \to \texttt{Favourite}\ f$. The surface-level constructor can then be represented as $\texttt{favourite}\ \texttt{refl}$, whilst $\texttt{subst}\ \texttt{Favourite}\ (\texttt{ext}\ p)\ \texttt{favourite}$ will compute to $\texttt{favourite}\ (\texttt{ext}\ p)$. With this representation of data types, the work done in this paper is just as necessary as before (modulo some details in the final proof), since we still need unification to solve the (telescopic) equations embedded in the constructors.

**Pattern matching with higher inductive types.** Our criterion makes it possible to do pattern matching on *regular* inductive families without assuming K. But HoTT also introduces the concept of *higher inductive types*, which can have non-trivial identity proofs between their constructors. This implies that in general they do not satisfy the injectivity, disjointness, or acyclicity properties. Luckily, the proof given in this paper is entirely *parametric* in the actual unification transitions that are used. So in order to allow pattern matching in a context with higher inductive types, we should start by limiting the unification algorithm further, for example, by cutting out the "no confusion" and "cycle" properties for types to which they don't apply.

As a second step, these principles can be replaced by type-specific solvers that exploit any extra structure which may be available. For example, the "no confusion" principle in this paper is very similar to the encode/decode technique used by Licata

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

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

and Shulman (2013) 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. Future research will have to show how much of the original pattern matching algorithm can be salvaged in this setting.

## 11 Conclusion

Dependent pattern matching is an important tool for writing dependently typed functions and proofs in a readable way, but so far it needed the K axiom to function. What this paper shows, is that there is no need to throw out the baby with the bath water: By carefully analysing where K is used, we can give a restricted formulation of dependent pattern matching that does not need it. We hope that this is enough to convince the HoTT community that pattern matching does not require K *an sich*, and maybe even helps in the creation of a practical language based on HoTT.

## References

Altenkirch, T. (2012) Without-K problem. Available at: `https://lists.chalmers.se/pipermail/agda/2012/004104.html`. On the Agda mailing list. (last accessed date 09/08/2016)

Augustsson, L. (1985) Compiling pattern matching. In *Functional Programming Languages and Computer Architecture: Nancy, France, September 16–19, 1985*, Jouannaud, J.-P. (ed), Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 368–381.

Barras, B., Corbineau, P., Grégoire, B., Herbelin, H. & Sacchini, J. L. (2009) A new elimination rule for the calculus of inductive constructions. In *Types for Proofs and Programs: International Conference, TYPES 2008 Torino, Italy, March 26–29, 2008 Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 32–48.

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)*, Matthes, R. and Schubert, A. (eds), Leibniz International Proceedings in Informatics (LIPIcs), vol. 26. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 107–128.

Boutillier, P. (2014) *De nouveaux outils pour Calculer avec des inductifs en Coq*. PhD Thesis, Université Paris-Diderot-Paris VII.

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

Cockx, J. (2014) Yet another way Agda –without-K is incompatible with univalence. Available at: `https://lists.chalmers.se/pipermail/agda/2014/006367.html`. On the Agda mailing list. (last accessed date 09/08/2016)

Cockx, J., Devriese, D. & Piessens, F. (2014) Pattern matching without K. In Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. New York, NY, USA: ACM, pp. 257–268.

Cockx, J., Devriese, D. & Piessens, F. (to appear) Unifiers as Equivalences: Proof-relevant unification of dependently typed data. In *Proceedings of the 21th ACM SIGPLAN International Conference on Functional Programming*. ACM.

https://doi.org/10.1017/S0956796816000174 Published online by Cambridge University Press

Cohen, C., Coquand, T., Huber, S. & Mörtberg, A. (2015) Cubical type theory: A constructive interpretation of the univalence axiom. `http://www.cse.chalmers.se/čoquand/cubicaltt.pdf` (last accessed date 09/08/2016)

Coquand, T. (1992) Pattern matching with dependent types. In *Proceedings of the 3rd Workshop on Types for Proofs and Program*, pp. 66–79.

Dagand, P.-E. (2013) *A Cosmology of Datatypes: Reusability and Dependent Types*. PhD Thesis, University of Strathclyde.

Danielsson, N.-A. (2013) Experiments related to equality. Available at: `http://www.cse.chalmers.se/ñad/repos/equality/`. Agda code. (last accessed date 09/08/2016)

de Moura, L., Kong, S., Avigad, J., van Doorn, F. & von Raumer, J. (2015) The lean theorem prover (system description). In Proceedings of 25th International Conference on Automated Deduction (CADE-25), Felty, P. Amy and Middeldorp, A. (eds). Cham: Springer International Publishing, pp. 378–388.

Devriese, D. & Piessens, F. (2011) On the bright side of type classes: Instance arguments in Agda. In ACM SIGPLAN International Conference on Functional Programming (ICFP), New York, NY, USA: ACM, pp. 143–155.

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, Huet, G. and Plotkin, G. (eds) pp. 213–230.

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*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 521–540.

Hofmann, M. & Streicher, T. (1994) The groupoid model refutes uniqueness of identity proofs. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pp. 208–212.

Jouannaud, J.-P. & Kirchner, C. (1990) *Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification* In *Computational Logic: Essays in Honor of Alan Robinson*, Lassez, J. L. (ed).

Kraus, N., Escardó, M., Coquand, T. & Altenkirch, T. (2013) Generalizations of Hedberg's theorem. In *Typed Lambda Calculi and Applications*, Hasegawa, M. (ed). Springer, pp. 173–188.

Kraus, N. & Sattler, C. (2015) On the hierarchy of univalent universes: U(n) is not n-truncated. *ACM Trans. Comput. Logic*. **16**(2), 18:1–18:12. New York, NY, USA: ACM.

Licata, D. (2011) Just kidding: Understanding identity elimination in homotopy type theory. Available at: `http://homotopytypetheory.org/2011/04/10/just-kidding-understanding-identity-elimination-in-homotopy-type-theory/`. (last accessed date 09/08/2016)

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

Luo, Z. (1994) *Computation and Reasoning: A Type Theory for Computer Science*, International Series of Monographs on Computer Science, vol. 11.

Mangin, C. & Sozeau, M. (2015) Equations for hereditary substitution in Leivant's predicative system F: A case study. In Proceedings of the 10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Cervesato, I. and Chaudhuri K. pp. 71–86.

Maranget, L. (2008) Compiling pattern matching to good decision trees. In Proceedings of the 2008 ACM SIGPLAN Workshop on ML. New York, NY, USA: ACM, pp. 35–46.

Martin-Löf, P. (1984) Intuitionistic Type Theory. *Naples: Bibliopolis* Number 1 in Studies in Proof Theory, vol. 76.

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

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: International Workshop, TYPES 2000 Durham, UK, December 8–12, 2000 Selected Papers*, Callaghan, P., Luo, Z., McKinna, J., and Pollack, R. (eds). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 197–216.

McBride, C. (2005) Epigram: Practical programming with dependent types. In *Advanced Functional Programming* Vene, V. and Uustalu. T. (eds). Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 130–170.

McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1), 69–111.

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 Berlin Heidelberg, pp. 186–200.

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

Norell, U., Abel, A. & Danielsson, N. A. (2012) Release notes for Agda 2 version 2.3.2. Available at: `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Version-2-3-2`. (last accessed date 09/08/2016)

Paulin-Mohring, C. (1993) Inductive definitions in the System Coq - rules and properties. In Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA '93. London, UK: Springer-Verlag, pp. 328–345.

Reed, J. (2013) Another possible without-K problem. Available at: `https://lists.chalmers.se/pipermail/agda/2013/005578.html`. On the Agda mailing list. (last accessed date 09/08/2016)

Sicard-Ramírez, A. (2013) –without-K option too restrictive?. Available at: `https://lists.chalmers.se/pipermail/agda/2013/005407.html`. On the Agda mailing list. (last accessed date 09/08/2016)

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

Sozeau, M. (2015) Coq support for HoTT. In *Workshop on Homotopy Type Theory / Univalent Foundations*.

The Coq development team. (2012) *The Coq Proof Assistant Reference Manual*. LogiCal Project. Available at: `http://coq.inria.fr`. Version 8.4.

The Univalent Foundations Program. (2013) *Homotopy Type Theory: Univalent Foundations of Mathematics*. Available at: `http://homotopytypetheory.org/book`, Institute for Advanced Study. (last accessed date 09/08/2016)