Draft of June 5, 2014

Pattern Matching Without K

Jesper Cockx Dominique Devriese Frank Piessens

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium.

firstname.lastname@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. The Agda language supports an experimental criterion to detect definitions by pattern matching that make use of the K axiom, but so far it lacked a formal correctness proof.

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). Our criterion both allows more good definitions than existing proposals, and solves a previously undetected problem in the 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. It also points the way to new forms of dependent pattern matching, for example on higher inductive types.

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

Keywords Dependent Pattern Matching, K Axiom, Homotopy Type Theory, Agda

1. Introduction

The case for dependent pattern matching. 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 $\texttt{half} : \mathbb{N} \to \mathbb{N}$ can be defined as

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

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) by pattern matching on its only constructor $\texttt{refl}$ of type $x \equiv x$:

$$\begin{array}{l} \texttt{trans} : (x\ y\ z : A) \to x \equiv y \to y \equiv z \to x \equiv z \\ \texttt{trans}\ x\ \lfloor x \rfloor\ \lfloor x \rfloor\ \texttt{refl}\ \texttt{refl} = \texttt{refl} \end{array} \tag{2}$$

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 $\texttt{refl} : x \equiv x$ forces x and y to be the same. Another example is the proof $\texttt{cong}$ that any function maps equal arguments to equal results:

$$\begin{array}{l} \texttt{cong} : (f : A \to B)(x\ y : A) \to x \equiv y \to f\ x \equiv f\ y \\ \texttt{cong}\ f\ x\ \lfloor x \rfloor\ \texttt{refl} = \texttt{refl} \end{array} \tag{3}$$

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. For example, let $\leq$ be the usual ordering on $\mathbb{N}$ defined as an inductive family (Dybjer 1991) defined by the two constructors $\texttt{lz}$ and $\texttt{ls}$:

$$\begin{array}{l} \texttt{lz} : (n : \mathbb{N}) \to \texttt{zero} \leq n \\ \texttt{ls} : (m\ n : \mathbb{N}) \to m \leq n \to \texttt{suc}\ m \leq \texttt{suc}\ n \end{array} \tag{4}$$

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

$$\begin{array}{l} \texttt{antisym} : (m\ n : \mathbb{N}) \to m \leq n \to n \leq m \to m \equiv n \\ \texttt{antisym}\ \lfloor \texttt{zero} \rfloor\ \lfloor \texttt{zero} \rfloor\ (\texttt{lz}\ \lfloor \texttt{zero} \rfloor)\ (\texttt{lz}\ \lfloor \texttt{zero} \rfloor) = \\ \qquad \texttt{refl} \\ \texttt{antisym}\ \lfloor \texttt{suc}\ m \rfloor\ \lfloor \texttt{suc}\ n \rfloor\ (\texttt{ls}\ m\ n\ x)\ (\texttt{ls}\ \lfloor n \rfloor\ \lfloor m \rfloor\ y) = \\ \qquad \texttt{cong}\ \texttt{suc}\ (\texttt{antisym}\ m\ n\ x\ y) \end{array} \tag{5}$$

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

```
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)))
```

Figure 1. This proof of the antisymmetry of $\leq$ is more complex than the proof by pattern matching (5) because it uses only the standard datatype eliminators (see Section 3.1) and the "no confusion" property of the natural numbers. No confusion can be constructed from the eliminator for $\mathbb{N}$ as well (see Section 3.4).

formally in Section 3.1, but Figure 1 already gives an alternative definition of `antisym` as an example using eliminators only. 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 cleverness for the construction of the *motive* (McBride 2002) of each eliminator, while this is done automatically in the definition by pattern matching. 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:

$$\begin{aligned} \mathrm{K} &: (P : a \equiv a \to \mathtt{Set}) \to \\ &\quad (p : P\ \mathtt{refl})(e : a \equiv a) \to P\ e \\ \mathrm{K}\ \ P\ p\ \mathtt{refl} &= p \end{aligned} \tag{6}$$

The K axiom is equivalent with the *uniqueness of identity proofs* principle (UIP), 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). For the next example, remember that in type theory there is no strict boundary between types and terms, so we can form equations between types as well, for example `Bool` $\equiv$ `Bool`. Given such an equation between types, we can coerce terms of the first type to the other using the function `coerce` $: A \equiv B \to A \to B$ (which can be constructed by pattern matching). Now we can use pattern matching to prove that coercing `true` by any proof of `Bool` $\equiv$ `Bool` results in `true`:

```
coerce-id : (e : Bool ≡ Bool) → coerce e true ≡ true
coerce-id refl = refl
```
(7)

This can be desugared to

```
coerce-id = λe. K (λe. coerce e true ≡ true) refl e
```
(8)

The K axiom is necessary to deal with reflexive equations such as `Bool` $\equiv$ `Bool` in this example.

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 a certain sense).

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 `ua` f of $A \equiv B$. Moreover, this proof satisfies `coerce` (`ua` f) $x = f\ x$. Univalence captures the common mathematical practice of informal reasoning "up to isomorphism" in a nice and formalized way. It also has a number of useful consequences, such as *functional extensionality*.

However, the univalence axiom is *incompatible* with dependent pattern matching. For example, we can construct a function `swap` : `Bool` $\to$ `Bool` such that `swap true` $=$ `false` and vice versa. This function is its own inverse, so by univalence it gives us a proof `ua swap` of `Bool` $\equiv$ `Bool` such that coercing `true` along this proof results in `false`. Together with the proof `coerce-id` (7), this leads to a proof of the absurdity `true` $\equiv$ `false`. This has forced people working on HoTT to avoid using pattern matching or risk unsoundness.

Avoiding K. The source of the incompatibility between univalence and dependent pattern matching is that pattern matching relies on the K axiom. 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. One attempt to achieve this is an option in Agda called –without-K (Norell et al. 2012). When enabled, Agda attempts to detect definitions by pattern matching that make use of the K axiom by means of a syntactic check. In theory, this option should 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. While 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.
- 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.
- 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 2.3.4, our implementation will replace the old version of –without-K.

- Finally, we give an idea how to make pattern matching without K even less restrictive by analyzing which types satisfy K without assuming it as an axiom. Future work is still needed to make this analysis more robust.

Overview. The rest of this paper is organized as follows. In section 2, we describe our criterion for pattern matching without K and compare our implementation with the current one in Agda. Section 3 contains 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. In section 4, we discuss how pattern matching without K can be made less restrictive. Finally, we discuss related work in section 5.

Supplementary material for this paper can be found online at `http://people.cs.kuleuven.be/~jesper.cockx/Without-K/`. This page contains the implementation of our criterion in Agda, Agda files containing the examples given in this paper, and Agda files illustrating parts of the proof in Section 3.

2. The criterion

A definition by pattern matching can be thought of as given by a number of case splits on the arguments. For example, the function $\mathtt{half} : \mathbb{N} \to \mathbb{N}$ in Definition 1 is defined by first doing a case split on the argument $n : \mathbb{N}$ – giving us two cases $n = \mathtt{zero}$ and $n = \mathtt{suc}\ m$ – and then another case split on m. For definitions like this one, each case split corresponds exactly to one application of the standard eliminator for $\mathbb{N}$, hence the K axiom is not needed.

Things get more complicated for an inductive family (Dybjer 1991) such as $\mathtt{Fin}\ n$, the canonical finite set of n elements, or $m \leq n$, 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 the possible cases. This unification algorithm depends crucially on the K axiom, so we have to restrict it in order to remove this dependence.

In this section, we first describe the unification algorithm used by Goguen et al. (2006). Next, we describe our restricted unification algorithm that does not depend on K. We also compare our criterion with the syntactic criterion for pattern matching without K in Agda. Finally, we give a short evaluation of our implementation.

2.1 Case splitting by unification of the indices

When checking a definition by pattern matching, 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 $m \leq n$ with constructors $\mathtt{lz}$ and $\mathtt{ls}$ as given in Definition 4. Suppose we want to do a case split on a variable of type $k \leq k$, then we have to decide for what kind of arguments the two constructors give a result of the form $k \leq k$. In the case of $\mathtt{lz}$, this is when both k and the argument n are equal to $\mathtt{zero}$, while for $\mathtt{ls}$, this is when the two arguments m and n are equal and $k = \mathtt{suc}\ n$.

In general, suppose we are case splitting on a variable $x : \mathtt{D}\ \bar{u}$ where $\mathtt{D}$ is an inductive family with indices $\bar{u}$ (we consider $\mathtt{D}$ to already be applied to its parameters, if any). Suppose $\mathtt{D}$ 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 σ such that $\bar{u}\sigma = \bar{v}_i\sigma$. A unification problem is represented as a list of equations $u_1 = v_{i1}, \ldots, u_n = v_{in}$, and the following five unification transitions are used to simplify the problem step by step:

Deletion: $x = x, \Theta \Rightarrow \Theta$

Solution: $x = t, \Theta \Rightarrow \Theta[x \mapsto t]$ (if x is not free in t)

Injectivity: $\mathtt{c}\ \bar{s} = \mathtt{c}\ \bar{t}, \Theta \Rightarrow \bar{s} = \bar{t}, \Theta$

Conflict: $\mathtt{c_1}\ \bar{s} = \mathtt{c_2}\ \bar{t}, \Theta \Rightarrow \bot$ (if $\mathtt{c_1} \neq \mathtt{c_2}$)

Cycle: $x = \mathtt{c}\ \bar{p}[x], \Theta \Rightarrow \bot$ (if $x \prec \mathtt{c}\ \bar{p}[x]$)

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 σ.

Negative success: Either the **conflict** or the **cycle** rule applies, meaning that there exist no unifiers.

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 pattern* $\emptyset$, marking that case splitting resulted in zero cases.[1] 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 σ_i constructed by unification. For example, a function $f : (k : \mathbb{N}) \to k \leq k \to P\ k$ can be defined by the following patterns:

$$\begin{aligned} f\ \lfloor\mathtt{zero}\rfloor\ (\mathtt{lz}\ \lfloor\mathtt{zero}\rfloor) &= \ldots \\ f\ \lfloor\mathtt{suc}\ n\rfloor\ (\mathtt{ls}\ n; \lfloor n\rfloor) &= \ldots \end{aligned} \tag{9}$$

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

2.2 Restricting the unification rules

Our criterion for pattern matching without K works by limiting 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 (while 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 for constructor forms, our modified version is only complete for *linear* constructor forms (i.e. ones where each variable occurs only once).

As a first example, our criterion allows the definition of the standard J-eliminator for the propositional equality (also known as the principle of *based path induction* in HoTT) by pattern matching:

$$\begin{aligned} \mathtt{J} &: (P : (b : A) \to a \equiv b \to \mathtt{Set}) \to \\ &\quad (p : P\ a\ \mathtt{refl})(b : A)(e : a \equiv b) \to P\ b\ e \\ \mathtt{J}\ \ P\ p\ \lfloor a\rfloor\ \mathtt{refl} &= p \end{aligned} \tag{10}$$

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.

[1] 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).

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.

$$\begin{aligned} &\mathrm{K} : (P : a \equiv a \to \mathtt{Set}) \to \\ &\quad (p : P\ \mathtt{refl})(e : a \equiv a) \to P\ e \\ &\mathrm{K}\quad P\ p\ \mathtt{refl} = p \end{aligned} \tag{11}$$

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:

$$\begin{aligned} &\mathtt{weakK} : (P : \mathtt{refl} \equiv_{a\equiv a} \mathtt{refl} \to \mathtt{Set}) \to \\ &\quad (p : P\ \mathtt{refl})(e : \mathtt{refl} \equiv_{a\equiv a} \mathtt{refl}) \to P\ \mathtt{refl} \\ &\mathtt{weakK}\quad P\ p\ \mathtt{refl} = p \end{aligned} \tag{12}$$

Like the regular K, this `weakK` does not follow from the standard rules of type theory and is incompatible with univalence (Kraus and Sattler 2013). However, since `refl` is a constructor without any arguments, it would be accepted if we did not have the second restriction.

2.3 Comparison with the syntactic criterion

So far, the only credible proposal of a criterion for pattern matching without K was the syntactic criterion 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 3. But we should also compare their generality, i.e. what kind of definitions are still allowed by each. The criterion currently 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). So the fact that our criterion is more restrictive in this case is actually a good thing.

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), while the second one is an index (so it must consist of constructors applied to free variables). In contrast, our criterion allows both variants because we look at the unifications that are performed instead of syntactical artefacts like the distinction between a parameter and an index. Similarly, Agda's syntactic criterion does not allow us to pattern match on an argument of type $n \leq n$ 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 parametrized data type. 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.

Unfortunately, our criterion still has some limitations. For example, when working with the $\leq$ relation on finite sets `Fin` n, we cannot pattern match on an argument of type $i \leq i$ 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 4, we discuss a possible solution to this problem.

2.4 Implementation and evaluation

Our new criterion for pattern matching without K has been implemented as a patch to Agda. We used it 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 11, 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 does not actually rely on the K axiom. For example, definitions involving case splitting on types such as $m \leq m$, $k \equiv l + m$, and $x \equiv f\ y$ are accepted. Another notable advantage of our criterion is that adding parameters to a data type will never change the validity of a definition by pattern matching. This is especially useful in Agda since module parameters are also considered to be parameters of the datatypes defined inside that module (Norell 2007, chapter 4). So with the syntactic criterion, moving a definition to another module can cause an error, but with our criterion this is no longer the case.

3. Eliminating pattern matching without K

In this section, we 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. Our proof follows the same general outline as the proof by Goguen et al. (2006), but there are two important differences:

- We work with the homogeneous propositional equality instead of the heterogeneous version. The reason is that the elimination rule they use for the heterogeneous equality is equivalent with K (McBride 2000), something we wish to avoid. Using the homogeneous equality also means that we have to work a little harder to express equality between two sequences of terms in the same telescope.
- Working with the homogeneous equality leads us very naturally to upgraded versions of the unification transitions given by Goguen et al. (2006), where the return type is dependent on the equality proof. The construction of these upgraded transitions will make clear why the two restrictions to the unification algorithm given in Section 2.2 are really needed.

The general idea of the proof is as follows. First, the definition by pattern matching is translated to a case tree. This translation is described in detail by Norell (2007), and we will not repeat it here. Each leaf node of the case tree corresponds to a clause $\mathtt{f}\ \bar{p} = e$, i.e. it defines $\mathtt{f}$ on arguments that match the pattern $\bar{p}$, and each internal node corresponds to a case split of $\bar{p}$ on some variable $x : \mathtt{D}\ \bar{u}$ into patterns $\bar{p}_1, \ldots, \bar{p}_n$. If we can assemble the definitions of $\mathtt{f}\ \bar{p}_1, \ldots, f\ \bar{p}_n$ into a definition of $\mathtt{f}\ \bar{p}$, then we can work backwards from the leaf nodes towards the root, ultimately obtaining a definition of $\mathtt{f}$ on arbitrary variables.

So we need to know how to assemble the definitions of $\mathtt{f}\ \bar{p}_1, \ldots, \mathtt{f}\ \bar{p}_n$ into a definition of $\mathtt{f}\ \bar{p}$. This assembly proceeds in two steps. First we apply a technique called *basic* $\mathtt{case_D}$*-analysis at* $\bar{u}; x$. This splits the problem into one subproblem for each constructor c_i of $\mathtt{D}$, and gives us proofs of the equations $\bar{u} = \bar{v}_i$ and $x = \mathtt{c}\ \bar{y}$. 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. So finally, we fill in the translated definition of $\mathtt{f}\ \bar{p}_i$ for each positive success.

In general there can be recursive calls to the function $\mathtt{f}$ in each clause $\mathtt{f}\ \bar{p} = e$. These recursive calls are required to be structurally recursive on some argument $x : \mathtt{D}\ \bar{u}$ of $\mathtt{f}$. It is important for the proof that the type of x in Δ is already a data type, not just the type of x in each of the clauses separately. This allows us to use well-founded recursion on $\mathtt{D}$ to obtain an inductive hypothesis H, asserting that $\mathtt{f}$ is already defined on arguments structurally smaller than x. This inductive hypothesis is then used to replace the recursive calls to $\mathtt{f}$ in e.

The challenge is then to construct all these techniques (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 3.1) and dependent pattern matching (Section 3.2). We continue by showing how the homogeneous propositional equality can be used to express equality of sequences (Section 3.3). 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 work with the homogeneous equality (Section 3.4). No confusion and acyclicity are subsequently used to construct the unification transitions as terms inside type theory (Section 3.5). Finally, all these tools are brought together for the translation of case trees to eliminators (Section 3.6).

3.1 Type theory

As our version of type theory, we use Luo's Unified Theory of Dependent Types (UTT) 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 UTT we use are summarized in Fig. 2.

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash f : (x : A) \to B \qquad x \notin FV(f)}{\lambda x.\ f\ x = f : (x : A) \to B}\ (\eta)$$

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

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

Contexts and substitutions. We use Greek capitals $\Gamma, \Delta, \ldots$ for contexts, capitals $T, U, \ldots$ for types, and small letters $t, u, \ldots$ for terms. A list of terms is indicated by a bar above the letter, for example $\bar{t}$. Contexts double as the type of such a list of terms, also called a *telescope*, so we can write for example $\bar{t} : \Gamma$ where $\Gamma = (m : \mathbb{N})(p : m \equiv \mathtt{zero})$ and $\bar{t} = \mathtt{zero}; \mathtt{refl}$. Note that the empty context ϵ is inhabited by the empty list $()$. The simultaneous substitution of the terms $\bar{t}$ for the variables in the context Γ is written as $[\Gamma \mapsto \bar{t}]$. We denote substitutions by small Greek letters $\sigma, \tau, \ldots$

Elimination operators. For any telescope Ξ, we define a Ξ-*elimination operator* (McBride 2002) to be any function with a type of the form

$$\begin{aligned}&(P : \Xi \to \mathtt{Set}_i) \to \\ &(m_1 : \Delta_1 \to P\ \bar{s}_1) \ldots (m_n : \Delta_n \to P\ \bar{s}_n) \to \\ &(\bar{t} : \Xi) \to P\ \bar{t}\end{aligned} \tag{13}$$

We call Ξ the *target*, P the *motive*, and $m_1, \ldots, m_n$ the *methods* of the elimination operator. The reader may think of a Ξ-elimination operator as a way to transform a 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 an arbitrary list of values $\bar{t}$ in the telescope Ξ. This original problem is transformed into n sub-problems given by each of the methods: the ith subproblem is to construct a result of type $P\ \bar{s}_i$ when given an arbitrary value satisfying telescope Δ_i. The elimination operator's type can be read as a function that

transforms solutions for the sub-problems into a solution for the original problem.

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*, 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

$$\begin{aligned}\texttt{elim}_{\mathbb{N}} : (P : \mathbb{N} \to \texttt{Set}_i) \to (m_{\texttt{zero}} : P\ \texttt{zero}) \to \\ (m_{\texttt{suc}} : (n : \mathbb{N}) \to P\ n \to P\ (\texttt{suc}\ n)) \to \\ (n : \mathbb{N}) \to P\ n\end{aligned} \tag{14}$$

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 Ξ of the indices and the constructors

$$\texttt{c}_\texttt{i} : \Delta_i \to (\Phi_{i1} \to \texttt{D}\ \bar{v}_{i1}) \to \ldots \to (\Phi_{in_i} \to \texttt{D}\ \bar{v}_{in_i}) \to \texttt{D}\ \bar{u}_i \tag{15}$$

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 $\texttt{D}$ is a $\bar{\texttt{D}}$-elimination operator with methods $m_1, \ldots, m_k$ where

$$\begin{aligned}m_i : (\bar{t} : \Delta_i) \to \\ (x_1 : \Phi_{i1} \to \texttt{D}\ \bar{v}_{i1})\ \ldots\ (x_{n_i} : \Phi_{in_i} \to \texttt{D}\ \bar{v}_{in_i}) \to \\ (h_1 : (\bar{s}_1 : \Phi_{i1}) \to P\ \bar{v}_{i1}\ (x_1\ \bar{s}_1)) \to \ldots \to \\ (h_{n_i} : (\bar{s}_{n_i} : \Phi_{in_i}) \to P\ \bar{v}_{in_i}\ (x_{n_i}\ \bar{s}_{n_i})) \to \\ P\ \bar{u}_i\ (\texttt{c}_\texttt{i}\ \bar{t}\ x_1\ \ldots\ x_{n_i})\end{aligned} \tag{16}$$

i.e. it is of the form

$$\begin{aligned}\texttt{elim}_{\texttt{D}} : (P : \bar{\texttt{D}} \to \texttt{Set}_i)(m_1 : \ldots) \ldots (m_k : \ldots) \to \\ (\bar{x} : \bar{\texttt{D}}) \to P\ \bar{x}\end{aligned} \tag{17}$$

where the types of $m_1, \ldots, m_k$ are as given above.

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 $\beta\eta$-conversion. 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 (1984). In UTT, it can be defined as an inductive family with two parameters $A : \texttt{Set}_i$ and $a : A$, one index $b : A$, and one constructor $\texttt{refl} : a \equiv a$. The standard eliminator for this datatype is exactly the J rule (10). Substitution by a propositional equality $\texttt{subst} : (P : A \to \texttt{Set}_i) \to x \equiv y \to P\ x \to P\ y$ 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 write e_* for $\texttt{subst}\ P\ e$ when P is clear from the context.

3.2 Definitions by pattern matching

A definition by pattern matching of a function $\texttt{f}$ consists of a number of equalities called *clauses*, which are of the form $\texttt{f}\ \bar{p} = t$ where $\bar{p}$ is a list of patterns and t is a term called the *right-hand side*. A *pattern* is a term or a list of terms that is built from only (fully applied) constructors and (non-applied) variables, which we call the pattern variables. In dependent pattern matching, patterns can also contain *inaccessible patterns*, which can occur when there is only one type-correct term possible in a given position. Like Norell (2007), we mark inaccessible patterns as $\lfloor t \rfloor$. For example, let $\texttt{Square}\ n$ be an inductive family with one index $n : \mathbb{N}$ and one constructor

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

Figure 3. A representation of the function $\texttt{half}$ by a case tree. At each internal node, the variable on which the case split is performed is underlined.

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

Figure 4. The structural order $\prec$ is used to check termination (Goguen et al. 2006).

$\texttt{sq} : (m : \mathbb{N}) \to \texttt{Square}\ m^2$. Then $\lfloor m^2 \rfloor\ (\texttt{sq}\ m)$ is a pattern of type $(n : \mathbb{N})(p : \texttt{Square}\ n)$. Any other pattern $\lfloor t \rfloor\ (\texttt{sq}\ m)$ would be ill-typed, so the use of an inaccessible pattern is justified. We also define an operation $\lceil \bar{p} \rceil$ taking a pattern $\bar{p}$ back to its underlying term.

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 $\texttt{f} : \Delta \to T$ are labeled by patterns of type Δ, where the label of the root node consists of variables only. Each internal node of a case tree corresponds to a case split, while each leaf node corresponds to a clause of the definition. An example of a case tree is given in Figure 3.

Using case trees has a number of advantages. First, the patterns at the leaves of a case tree always form a covering, hence case trees guarantee completeness. Secondly, they give an efficient method to evaluate functions defined by pattern matching. Thirdly 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).

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 4 . For functions with multiple arguments, the function should be structurally recursive on one of its arguments, i.e. there should be some k such that $s_k \prec p_k$ for each clause $\texttt{f}\ \bar{p} = t$ and each recursive call $\texttt{f}\ \bar{s}$ in t.

3.3 Homogeneous telescopic equality

There is a reason why it is hard to see where exactly the K axiom is used in the translation from pattern matching to eliminators by Goguen et al. (2006): they do not use the axiom directly, but instead depend on the heterogeneous propositional equality. The heterogeneous equality allows the formation of equalities between terms of different types, but still only allows a proof when the types are in fact the same. This heterogeneous equality is convenient for expressing equality between sequences of data in a given telescope. 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. So instead we work with the *homogeneous* propositional equality and the standard J eliminator.

Working with homogeneous equality also means we have to work a little harder to express equality of two sequences of terms in the same telescope. We define telescopic equality $\bar{s} \equiv \bar{t}$ inductively

on the length of the telescope as follows:

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

where $e_*\ (s_1;\ldots;s_n) := (e_*\ s_1;\ldots;e_*\ s_n)$. Note that the substitution e_* is needed to make the equation between $\bar{s}$ and $\bar{t}$ again homogeneous. 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 $\overline{\texttt{refl}} : \bar{t} \equiv \bar{t}$ as $\texttt{refl};\ldots;\texttt{refl}$. We also have the telescopic eliminator

$$\begin{array}{l} \bar{\mathtt{J}} : (P : (\bar{s} : \Delta) \to \bar{r} \equiv \bar{s} \to \mathtt{Set}_i) \\ \quad P\ \bar{r}\ \overline{\texttt{refl}} \to (\bar{s} : \Delta) \to (\bar{e} : \bar{r} \equiv \bar{s}) \to P\ \bar{s}\ \bar{e} \end{array} \tag{19}$$

It is defined by eliminating the equations $\bar{e}$ from left to right using J. Each elimination of an equation $e_i : r_i \equiv s_i$ fills in `refl` for all occurrences of e_i, allowing the next equations to reduce and in particular ensuring that the following equation is of the correct form. Telescopic substitution $\overline{\texttt{subst}}$ is defined by dropping the dependence of Φ on $\bar{r} \equiv \bar{s}$ in the definition of $\bar{\mathtt{J}}$. Again, we write $\bar{e}_*$ for $\overline{\texttt{subst}}\ P\ \bar{e}$ when P is clear from the context. A formalization of homogeneous telescopic equality and the constructions in this section can be found in the file `TelescopicEquality.agda` in the supplementary material.

3.4 A few homogeneous constructions on constructors

McBride et al. (2006) developed tools for working with inductive families of datatypes: case analysis, recursion, no confusion (subsuming both injectivity and disjointness), and acyclicity. In this section, we present these rules adapted to work with homogeneous instead of heterogeneous equality. We refer to the appendix for the actual construction of these tools, as the differences with the work of McBride et al. are minor and rather technical. A computer-checked version of these constructions for some concrete data types (binary trees, dependent sums, finite sets, the identity type, and indexed containers) can be found in the file `ConstructionsOnConstructors.agda` in the supplementary material. For the rest of this section, let $\mathtt{D} : \Xi \to \mathtt{Set}_i$ be an inductive family.

Case analysis $\mathtt{case}_\mathtt{D}$ is a weakened version of the standard eliminator $\mathtt{elim}_\mathtt{D}$ that we get by dropping the inductive hypotheses of the methods. For example, $\mathtt{case}_\mathbb{N}$ has type

$$\begin{array}{l} (P : \mathbb{N} \to \mathtt{Set}_i) \to (m_{\mathtt{zero}} : P\ \mathtt{zero}) \to \\ (m_{\mathtt{suc}} : (n : \mathbb{N}) \to P\ (\mathtt{suc}\ n)) \to (n : \mathbb{N}) \to P\ n \end{array} \tag{20}$$

Recursion is given in two levels. First, for $x : \mathtt{D}\ \bar{u}$, $\mathtt{Below}_\mathtt{D}\ P\ \bar{u}\ x$ is a tuple type that is inhabited whenever $P\ \bar{v}\ y$ holds for all $y : \mathtt{D}\ \bar{v}$ which are structurally smaller than $x : \mathtt{D}\ \bar{u}$. For example for $\mathbb{N}$, we have $\mathtt{Below}_\mathbb{N}\ P\ \mathtt{zero} = \top$ (the unit type) and $\mathtt{Below}_\mathbb{N}\ P\ (\mathtt{suc}\ n) = \mathtt{Below}_\mathbb{N}\ P\ n \times P\ n$. Secondly, the helper function $\mathtt{below}_\mathtt{D}$ constructs this tuple:

$$\begin{array}{l} \mathtt{below}_\mathtt{D} : (P : (\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Set}_i) \to \\ \qquad ((\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Below}_\mathtt{D}\ P\ \bar{x} \to P\ \bar{x}) \to \\ \qquad (\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Below}_\mathtt{D}\ P\ \bar{x} \end{array} \tag{21}$$

Finally,

$$\begin{array}{l} \mathtt{rec}_\mathtt{D} : (P : (\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Set}_i) \to \\ \qquad ((\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Below}_\mathtt{D}\ P\ \bar{x} \to P\ \bar{x}) \to \\ \qquad (\bar{x} : \bar{\mathtt{D}}) \to P\ \bar{x} \end{array} \tag{22}$$

is used for well-founded recursion over values of type D.

No confusion is also given in two levels. First, $\mathtt{NoConfusion}_\mathtt{D} : \bar{\mathtt{D}} \to \bar{\mathtt{D}} \to \mathtt{Set}_d$ is a type such that

$$\begin{array}{l} \mathtt{NoConfusion}_\mathtt{D}\ (\bar{u};c\ \bar{s})\ (\bar{v};c\ \bar{t}) = \bar{s} \equiv \bar{t} \\ \mathtt{NoConfusion}_\mathtt{D}\ (\bar{u};c\ \bar{s})\ (\bar{v};c'\ \bar{t}) = \bot\ (\text{when } c \neq c') \end{array} \tag{23}$$

Secondly, we construct

$$\mathtt{noConf}_\mathtt{D} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \bar{x} \equiv \bar{y} \to \mathtt{NoConfusion}_\mathtt{D}\ \bar{x}\ \bar{y} \tag{24}$$

We also construct an inverse

$$\mathtt{noConf}_\mathtt{D}{}^{-1} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \mathtt{NoConfusion}_\mathtt{D}\ \bar{x}\ \bar{y} \to \bar{x} \equiv \bar{y} \tag{25}$$

and give a proof $\mathtt{isLeftInv}_\mathtt{D}$ that $(\mathtt{noConf}_\mathtt{D}{}^{-1}\ \bar{x}\ \bar{y}) \circ (\mathtt{noConf}\ \bar{x}\ \bar{y})$ is the identity on $\bar{x} \equiv \bar{y}$.[2] The need for this inverse will become clear when we construct the unification transitions in Section 3.5.

Acyclicity is yet again given in two levels. First, $\bar{x} \not< \bar{y}$ is defined as a tuple type stating that $\bar{x} : \bar{\mathtt{D}}$ is not structurally smaller than $\bar{y} : \bar{\mathtt{D}}$. For example, $x \not< 2 = (x \not\equiv 0) \times (x \not\equiv 1)$. Secondly, $\mathtt{noCycle}_\mathtt{D} : (\bar{x}\ \bar{y} : \bar{\mathtt{D}}) \to \bar{x} \equiv \bar{y} \to \bar{x} \not< \bar{y}$ states that no term can be structurally smaller than itself.

Basic analysis. Note that $\mathtt{elim}_\mathtt{D}$, $\mathtt{case}_\mathtt{D}$, and $\mathtt{rec}_\mathtt{D}$ are all $\bar{\mathtt{D}}$-elimination operators, i.e. for a motive $P : \bar{\mathtt{D}} \to \mathtt{Set}_j$ they return something of type $(\bar{u} : \Xi)(x : \mathtt{D}\ \bar{u}) \to P\ (\bar{u};x)$. However, we often need a return type where the indices $\bar{u}$ of x are more specialized, for example to construct a function of type $(k : \mathbb{N})(y : k \leq \mathtt{zero}) \to \mathtt{zero} \equiv k$. McBride (2002) solves this problem by adding the constraints on the indices as additional arguments to the motive P, and filling in $\overline{\texttt{refl}}$ as soon as the constraints are satisfied. This technique is called *basic analysis*. In the example above, the basic $\mathtt{case}_\leq$-analysis of $\mathtt{zero} \equiv k$ at $k;\mathtt{zero};y$ has type

$$\begin{array}{l} (m_1 : (m : \mathbb{N})(k : \mathbb{N})(y : k \leq \mathtt{zero}) \to \\ \quad (\mathtt{zero};m;\mathtt{lz}\ m) \equiv (k;\mathtt{zero};y) \to \mathtt{zero} \equiv k) \to \\ (m_2 : (m\ n : \mathbb{N})(x : m \leq n)(k : \mathbb{N})(y : k \leq \mathtt{zero}) \to \\ \quad (\mathtt{suc}\ m;\mathtt{suc}\ n;\mathtt{ls}\ m\ n\ x) \equiv (k;\mathtt{zero};y) \to \mathtt{zero} \equiv k) \to \\ (k : \mathbb{N})(y : k \leq \mathtt{zero}) \to \mathtt{zero} \equiv k \end{array} \tag{26}$$

Note that applying $\mathtt{case}_\leq$ directly to $y : k \leq \mathtt{zero}$ would lead to loss of the information that the second index of y is `zero`, thus leaving us unable to provide m_1 and m_2.

In general, let `elim` be any Ξ-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 `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 $\overline{\texttt{refl}}$ for the proof of $\bar{s} \equiv \bar{t}$ gives us the *basic `elim`-analysis of* Φ *at* $\bar{t}$:

$$\begin{array}{l} \lambda m_1;\ldots;m_n;\bar{x}. \\ \quad \mathtt{elim}\ (\lambda\bar{s}.\ \Delta \to \bar{s} \equiv \bar{t} \to \Phi)\ m_1\ \ldots\ m_n\ \bar{t}\ \bar{x}\ \overline{\texttt{refl}} \end{array} \tag{27}$$

which is of type

$$\begin{array}{l} (m_1 : \Delta_1\Delta \to \bar{s}_1 \equiv \bar{t} \to \Phi) \to \ldots \to \\ (m_n : \Delta_n\Delta \to \bar{s}_n \equiv \bar{t} \to \Phi) \to \Delta \to \Phi \end{array} \tag{28}$$

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

3.5 Unification without K

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

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

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

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

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

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

Figure 5. 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 Φ has an additional dependence on the equality proof. While 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'` (33), `conflict'` (34), and `cycle'` (35).

process inside of type theory itself. In order to do this, we use the "no confusion" and "no cycle" properties from the previous section. The unification transitions are given in Figure 5. A computer-checked construction of them for some concrete data types can be found in the file `Unification.agda` in the supplementary material. 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 context $\Xi = (a : A)(b : B\ a)$ and a Ξ-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:

$$\Delta \to (e_a : a \equiv a') \to (e_a)_*\ b \equiv b' \to T \tag{29}$$

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 Φ on the equality proofs is caused by the need to use substitution in the definition of homogeneous telescopic equality. Intuitively, it is not surprising that not assuming UIP leads us to consider identity proofs relevant!

The first thing we want to point out about Figure 5 is the lack of a `deletion` transition. The non-dependent version of `deletion` given by Goguen et al. (2006) has type

$$(\Phi : \texttt{Set}_i) \to (m : \Phi) \to (e : x_0 \equiv x_0) \to \Phi \tag{30}$$

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

$$
\begin{aligned}
\texttt{deletion} &: (\Phi : (e : x_0 \equiv x_0) \to \texttt{Set}_i) \to \\
&\quad (m : \Phi\ \texttt{refl}) \to \\
&\quad (e : x_0 \equiv x_0) \to \Phi\ e
\end{aligned} \tag{31}
$$

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.

A second point of interest in Figure 5 is the type of Φ 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 $\texttt{c}\ \bar{s}$ and $\texttt{c}\ \bar{t}$. But this does not correspond exactly to the **injectivity** rule from Section 2.1. Rather, we need a more specialized version of `injectivity` where the indices $\bar{u}_s$ and $\bar{u}_t$ are already definitionally equal:

$$
\begin{aligned}
\texttt{injectivity}_\texttt{bad} &: (\Phi : (e : \texttt{c}\ \bar{s} \equiv \texttt{c}\ \bar{t}) \to \texttt{Set}_i) \to \\
&\quad (m : (\bar{e} : \bar{s} \equiv \bar{t}) \to \Phi\ ???)) \to \\
&\quad (e : \texttt{c}\ \bar{s} \equiv \texttt{c}\ \bar{t}) \to \Phi\ e
\end{aligned} \tag{32}
$$

However, unlike `injectivity` such a function can *not* be constructed from $\texttt{noConf}_\texttt{D}$. This is because in order to fill in the question marks, we need a function $g : \bar{s} \equiv \bar{t} \to \texttt{c}\ \bar{s} \equiv \texttt{c}\ \bar{t}$ such that we can prove $g\ (\texttt{noConf}_\texttt{D}\ (\bar{u}_s; \texttt{c}\ \bar{s})\ (\bar{u}_t; \texttt{c}\ \bar{t})\ \overline{\texttt{refl}}\ e) \equiv e$ 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 $\texttt{noConf}_\texttt{D}$ is the following:

$$
\begin{aligned}
\texttt{injectivity'} &: (\Phi : (\bar{e} : \bar{u}; \texttt{c}\ \bar{s} \equiv \bar{u}; \texttt{c}\ \bar{t}) \to \texttt{Set}_i) \to \\
&\quad (m : (\bar{e} : \bar{s} \equiv \bar{t}) \to \\
&\qquad \Phi\ (\texttt{noConf}_\texttt{D}^{-1}\ (\bar{u}; \texttt{c}\ \bar{s})\ (\bar{u}; \texttt{c}\ \bar{t})\ \bar{e})) \to \\
&\quad (e : \texttt{c}\ \bar{s} \equiv \texttt{c}\ \bar{t}) \to \Phi\ \overline{\texttt{refl}}\ e
\end{aligned} \tag{33}
$$

This rule is simply a specialized version of the `injectivity` rule in Figure 5. However, there is still a problem with this rule. Suppose we want to use it to construct a function of type $(e : \texttt{c}\ \bar{s} \equiv \texttt{c}\ \bar{t}) \to \Phi'\ e$ where $\Phi' : \texttt{c}\ \bar{s} \equiv \texttt{c}\ \bar{t} \to \texttt{Set}_i$, and we want to apply `injectivity'`. Then we need to find $\Phi : \bar{u}; \texttt{c}\ \bar{s} \equiv \bar{u}; \texttt{c}\ \bar{t} \to \texttt{Set}_i$ such that $\Phi\ \overline{\texttt{refl}}\ e = \Phi'\ e$ for arbitrary $e : \texttt{c}\ \bar{s} \equiv \texttt{c}\ \bar{t}$. This is problematic because we cannot eliminate the equations $\bar{u} \equiv \bar{u}$ in general without using the K axiom. This is the reason for the second restriction on the unification algorithm in our criterion, namely that the indices $\bar{u}$ should be *self-unifiable*. This condition guarantees that we can construct Φ from Φ' by applying the unification transitions used in the self-unification of $\bar{u}$ by applying *specialization by unification* (see below).

At first sight, the **conflict** and **cycle** rule suffer from the same problem as the **injectivity** rule because their motive Φ 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 $(e : \texttt{c}_1\ \bar{s} \equiv \texttt{c}_2\ \bar{t}) \to \Phi'\ e$. First we apply `conflict` with $\Phi = \lambda\bar{e}.\ \bot$, giving us a function of type $(\bar{e} : \bar{u}; \texttt{c}_1\ \bar{s} \equiv \bar{u}; \texttt{c}_2\ \bar{t}) \to \bot$. Filling in $\overline{\texttt{refl}}$ for the equations $\bar{u} \equiv \bar{u}$ gives us $(e : \texttt{c}_1\ \bar{s} \equiv \texttt{c}_2\ \bar{t}) \to \bot$. Now by $\bot$-elimination, we also get a function $(e : \texttt{c}_1\ \bar{s} \equiv \texttt{c}_2\ \bar{t}) \to \Phi'\ e$. This gives us the following rule:

$$
\begin{aligned}
\texttt{conflict'} &: (\Phi : (e : \texttt{c}_1\ \bar{s} \equiv \texttt{c}_2\ \bar{t}) \to \texttt{Set}_i) \to \\
&\quad (e : \texttt{c}_1\ \bar{s} \equiv \texttt{c}_2\ \bar{t}) \to \Phi\ e
\end{aligned} \tag{34}
$$

Analogously we can construct a function

$$
\begin{aligned}
\texttt{cycle'} &: (\Phi : (e : x \equiv \texttt{c}\ \bar{s}[x]) \to \texttt{Set}_i) \to \\
&\quad (e : x \equiv \texttt{c}\ \bar{s}[x]) \to \Phi\ \bar{e}
\end{aligned} \tag{35}
$$

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

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 $\mathtt{case}_{\mathtt{D}}$-analysis), we may seek to construct an inhabitant of this type, called a *specializer*, by exhaustively iterating the unification transitions as applicable. In case of a positive success, a specializer is found, given some $m : \Delta' \to T\sigma$ where $\sigma : \Delta' \to \Delta$ is a substitution. In the case of a negative success, a specializer is found without any additional assumptions.

The `solution` rule removes one variable from Δ, while `injectivity'` keeps it the same. Hence in the case of a positive success we have $\Delta' \subseteq \Delta$, and σ is idempotent. So for any $\bar{t} : \Delta$, we can define an inverse $\sigma^{-1}[\bar{t}] : \Delta'$ by selecting the variables of Δ' from Δ. If specialization by unification delivers a specializer s satisfying

$$(m : \Delta'.T\sigma) \vdash s : \Delta \to \bar{u} \equiv \bar{v} \to T \tag{36}$$

and $\bar{t} : \Delta$ is such that $\bar{u}[\Delta \mapsto \bar{t}] = \bar{v}[\Delta \mapsto \bar{t}]$, then we have $s\ \bar{t}\ \overline{\mathtt{refl}} \rightsquigarrow m\ \sigma^{-1}[\bar{t}]$. It is clear why this holds for `solution`, it also holds for `injectivity'` since both `noConf` and `isLeftInv` map $\overline{\mathtt{refl}}$ to $\overline{\mathtt{refl}}$, hence `injectivity'` $\Phi\ m\ \overline{\mathtt{refl}} \rightsquigarrow m\ \mathtt{refl}$.

3.6 From case trees to eliminators

Now we use the tools described in the previous sections to translate a function $\mathtt{f} : (\bar{t} : \Delta) \to T$ given by a structurally recursive case tree to another one $\mathtt{f'} : (\bar{t} : \Delta) \to T$ constructed from eliminators only. As a running example, let $\mathtt{f} = \mathtt{antisym}$ from Definition (5). For this example, we have $\Delta = (m\ n : \mathbb{N})(x : m \le n)(y : n \le m)$ and $T = m \equiv n$. Define $\{e\}_{\mathtt{f}}^{\mathtt{f'}}$ by replacing all occurrences of `f` by `f'` in e. Then we have moreover that `f'` satisfies $\mathtt{f'}\ \bar{t} \rightsquigarrow^* \{u\}_{\mathtt{f}}^{\mathtt{f'}}$ whenever $\mathtt{f}\ \bar{t} \rightsquigarrow u$, i.e. it has the same reduction behaviour as `f`.

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

$$\lambda m^s; \bar{t}.\ \mathtt{rec}_{\mathtt{D}}\ P\ m^s\ (\bar{v}; t_j)\ \bar{t}\ \overline{\mathtt{refl}} \tag{37}$$

which has type

$$\begin{aligned}&(m^s : (\bar{x} : \bar{\mathtt{D}}) \to \mathtt{Below}_{\mathtt{D}}\ P\ \bar{x} \to P\ \bar{x}) \to \\ &(\bar{t} : \Delta) \to T\end{aligned} \tag{38}$$

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

$$\lambda\bar{x}; H; \bar{t}; \bar{e}.\ \bar{\mathtt{J}}\ (\lambda\bar{x}; \bar{e}.\ \mathtt{Below}_{\mathtt{D}}\ P\ \bar{x} \to T)\ (m\ \bar{t})\ (\mathtt{sym}\ \bar{e})\ H \tag{39}$$

where $\mathtt{sym} : \bar{x} \equiv \bar{y} \to \bar{y} \equiv \bar{x}$. For any $\bar{t} : \Delta$, we have

$$m^s\ (\bar{v}; t_j)\ H\ \bar{t}\ \overline{\mathtt{refl}} \rightsquigarrow m\ \bar{t}\ H \tag{40}$$

We will define `f'` as

$$\lambda\bar{t}.\ \mathtt{rec}_{\mathtt{D}}\ P\ m^s\ (\bar{v}; t_j)\ \bar{t}\ \overline{\mathtt{refl}} : (\bar{t} : \Delta) \to T \tag{41}$$

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 $\mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)$. Also note that

$$\begin{aligned}&\mathtt{f'}\ \bar{t} \rightsquigarrow^* \mathtt{rec}_{\mathtt{D}}\ P\ m^s\ (\bar{v}; t_j)\ \bar{t}\ \overline{\mathtt{refl}} \\ &\quad\rightsquigarrow^* m^s\ (\bar{v}; t_j)\ (\mathtt{below}_{\mathtt{D}}\ P\ m^s\ (\bar{v}; t_j))\ \bar{t}\ \overline{\mathtt{refl}} \\ &\quad\rightsquigarrow^* m\ \bar{t}\ (\mathtt{below}_{\mathtt{D}}\ P\ m^s\ (\bar{v}; t_j))\end{aligned} \tag{42}$$

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 context Θ and we wish to construct $m : \Theta \to \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau \to T\tau$ where $\tau = [\Delta \mapsto \lceil\bar{p}\rceil]$. Note that we have $\Theta = \Delta$ at the root node. There are three cases:

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

$$\begin{aligned}&\ldots \to \\ &(m_{\mathtt{c}} : (\bar{s} : \Delta_{\mathtt{c}}) \to \Theta \to \bar{u}_s; \mathtt{c}\ \bar{s} \equiv \bar{v}_y; y \to \\ &\quad \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau \to T\tau) \to \\ &\ldots \to \\ &\quad \Theta \to \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau \to T\tau\end{aligned} \tag{43}$$

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

$$\begin{aligned}&(m_{\mathtt{1z}} : (k\ m\ n : \mathbb{N})(x : m \le n)(y : n \le m) \to \\ &\qquad (\mathtt{zero}; k; \mathtt{1z}\ k) \equiv (m; n; x) \to \\ &\qquad \mathtt{Below}\ P\ m\ n\ x \to m \equiv n) \\ &(m_{\mathtt{1s}} : (k\ l : \mathbb{N})(u : k \le l) \to \\ &\qquad (m\ n : \mathbb{N})(x : m \le n)(y : n \le m) \to \\ &\qquad (\mathtt{suc}\ k; \mathtt{suc}\ l; \mathtt{1s}\ k\ l\ u) \equiv (m; n; x) \to \\ &\qquad \mathtt{Below}\ P\ m\ n\ x \to m \equiv n) \\ &(m\ n : \mathbb{N})(x : m \le n)(y : n \le m) \to \\ &\mathtt{Below}\ P\ m\ n\ x \to m \equiv n\end{aligned} \tag{44}$$

To construct the methods $m_{\mathtt{c}}$, we apply specialization by unification on the equations $\bar{u}_s; \mathtt{c}\ \bar{s} \equiv \bar{v}_y; y$, which we know will succeed by definition of a valid case tree. For the method $m_{\mathtt{1z}}$ above, the first step is to apply `solution` to the equation $\mathtt{zero} \equiv m$, simplifying the goal type to

$$\begin{aligned}&m'_{\mathtt{1z}} : (k\ n : \mathbb{N})(x : \mathtt{zero} \le n)(y : n \le \mathtt{zero}) \to \\ &\qquad (k; \mathtt{1z}\ k) \equiv (n; x) \to \\ &\qquad \mathtt{Below}\ P\ \mathtt{zero}\ n\ x \to \mathtt{zero} \equiv n\end{aligned} \tag{45}$$

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

$$\begin{aligned}&m_{\mathtt{1z;1s}} : (k\ l : \mathbb{N})(u : k \le l)(y : \mathtt{suc}\ k \le \mathtt{zero}) \\ &\qquad (\mathtt{suc}\ l; \mathtt{1s}\ k\ l\ u) \equiv (\mathtt{zero}; y) \to \\ &\qquad \mathtt{Below}_{\le}\ P\ \mathtt{zero}\ (\mathtt{suc}\ k)\ (\mathtt{1z}\ (\mathtt{suc}\ k)) \to \\ &\qquad \mathtt{zero} \equiv \mathtt{suc}\ k\end{aligned} \tag{46}$$

For each `c` with positive success, we have to deliver a

$$m'_{\mathtt{c}} : \Theta' \to \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau\sigma \to T\tau\sigma \tag{47}$$

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 `c` gives us exactly such a function. For $m_{\mathtt{1z}}$, the goal type becomes

$$\begin{aligned}&m'''_{\mathtt{1z}} : (k : \mathbb{N})(y : k \le \mathtt{zero}) \to \\ &\qquad \mathtt{Below}\ P\ \mathtt{zero}\ k\ (\mathtt{1z}\ k) \to \mathtt{zero} \equiv k\end{aligned} \tag{48}$$

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

For any $\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2 : \Theta_1(y : \mathtt{D'}\ \bar{v}_y)\Theta_2$, we have

$$\begin{aligned} m\ (\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2) &\rightsquigarrow^* m_{\mathtt{c}}\ \bar{s}\ (\bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2)\ \overline{\mathtt{refl}} \\ &\rightsquigarrow^* m'_{\mathtt{c}}\ \sigma^{-1}[\bar{s}; \bar{t}_1; \mathtt{c}\ \bar{s}; \bar{t}_2] \end{aligned} \tag{49}$$

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 `f`. In our example, the goal type for the second leaf node is

$$\begin{aligned} m_2 : (k\ l : \mathbb{N})(u : k \leq l)(v : l \leq k) \rightarrow \\ \mathtt{Below}_{\leq}\ P\ (\mathtt{suc}\ k)\ (\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ u) \rightarrow \\ \mathtt{suc}\ k \equiv \mathtt{suc}\ l \end{aligned} \tag{50}$$

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 $H : \mathtt{Below}_{\mathtt{D}}\ P\ (\bar{v}; t_j)\tau$. So consider a recursive call $\mathtt{f}\ \bar{r}$ in e_i. Since `f` is structurally recursive, we have $r_j \prec \lceil p_{ij} \rceil$ where $r_j : D\ \bar{w}$. By construction of $\mathtt{Below}_{\mathtt{D}}$, we have a projection π such that $\pi\ H : (\bar{t} : \Delta) \rightarrow \bar{w}; r_j \equiv \bar{v}; t_j \rightarrow T$. 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

$$\begin{aligned} \pi_1\ H : (m\ n : \mathbb{N})(x : m \leq n)(y : n \leq m) \rightarrow \\ (k; l; u) \equiv (m; n; x) \rightarrow m \equiv n \end{aligned} \tag{51}$$

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 $H = \mathtt{below}_{\mathtt{D}}\ P\ m^s\ (\bar{v}; t_j)$, we get

$$\begin{aligned} &\pi\ (\mathtt{below}_{\mathtt{D}}\ P\ m^s\ (\bar{v}; t_j))\ \bar{r}\ \overline{\mathtt{refl}} \\ &\rightsquigarrow^* m^s\ (\bar{w}; r_j)\ (\mathtt{below}_{\mathtt{D}}\ P\ m^s\ (\bar{w}; r_j))\ \bar{r}\ \overline{\mathtt{refl}} \\ &\rightsquigarrow^* m\ \bar{r}\ (\mathtt{below}_{\mathtt{D}}\ P\ m^s\ (\bar{w}; r_j)) = \mathtt{f'}\ \bar{r} \end{aligned} \tag{52}$$

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

For each clause

$$\mathtt{f}\ \bar{p}_i = e_i \tag{53}$$

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

$$\begin{aligned} \mathtt{f'}\ \lceil \bar{p}_i \rceil &\rightsquigarrow^* m\ \lceil \bar{p}_i \rceil\ (\mathtt{below}_{\mathtt{D}}\ P\ m^s\ \bar{u}\ \lceil p_{ij} \rceil) \\ &\rightsquigarrow^* m_{\mathtt{c}}\ \ldots \text{ (working our way down the case tree)} \\ &\rightsquigarrow^* m_i\ \bar{s}\ (\mathtt{below}_{\mathtt{D}}\ P\ m^s\ \bar{u}\ \lceil p_{ij} \rceil) \\ &\rightsquigarrow^* e'_i[H \mapsto \mathtt{below}_{\mathtt{D}}\ P\ m^s\ \bar{u}\ \lceil p_{ij} \rceil] \\ &\rightsquigarrow^* \{e_i\}_{\mathtt{f}}^{\mathtt{f'}} \end{aligned} \tag{54}$$

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

4. Making pattern matching without K less restrictive

In Section 2.3, 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 $\leq$ indexed over finite sets $\mathtt{Fin}\ n$, and we try to unify two successors in the same finite set. The problem $\mathtt{fs}\ n\ x = \mathtt{fs}\ n\ y$ requires solving $n = n$, but then we get stuck because we cannot use **deletion**. It can be proven that K 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.

$$\begin{aligned} &\mathrm{K}_{\mathbb{N}} : (n : \mathbb{N})(P : n \equiv n \rightarrow \mathtt{Set}) \rightarrow \\ &\quad P\ \mathtt{refl} \rightarrow (e : n \equiv n) \rightarrow P\ e \\ &\mathrm{K}_{\mathbb{N}}\ \ \mathtt{zero}\ P\ p\ \mathtt{refl} = p \\ &\mathrm{K}_{\mathbb{N}}\ \ (\mathtt{suc}\ n)\ P\ p\ e = \mathtt{subst}\ P\ (\mathtt{add\text{-}drop}\ e) \\ &\qquad\qquad (\mathrm{K}_{\mathbb{N}}\ n\ (P \circ \mathtt{add})\ p\ (\mathtt{drop}\ e)) \end{aligned}$$

where

$$\begin{aligned} \mathtt{add} &: n \equiv n \rightarrow \mathtt{suc}\ n \equiv \mathtt{suc}\ n \\ \mathtt{add} &= \mathtt{noConf}_{\mathbb{N}}^{-1}\ (\mathtt{suc}\ n)\ (\mathtt{suc}\ n) \\ \\ \mathtt{drop} &: \mathtt{suc}\ n \equiv \mathtt{suc}\ n \rightarrow n \equiv n \\ \mathtt{drop} &= \mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ n)\ (\mathtt{suc}\ n) \\ \\ \mathtt{add\text{-}drop} &: (e : \mathtt{suc}\ n \equiv \mathtt{suc}\ n) \rightarrow \mathtt{add}\ (\mathtt{drop}\ e) \equiv e \\ \mathtt{add\text{-}drop} &= \mathtt{isLeftInv}_{\mathbb{N}}\ (\mathtt{suc}\ n)\ (\mathtt{suc}\ n) \end{aligned}$$

Figure 6. 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 $\mathtt{noConf}$, $\mathtt{noConf}^{-1}$ and $\mathtt{isLeftInv}$ constructed from eliminators in the appendix, but we could define these functions using pattern matching as well.

Looking back at the construction of the unification transitions in Section 3.5, 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 6 for a proof of this fact), so it would be fine to use **deletion** on $n = n$ when $n : \mathbb{N}$. This would already solve the problem described above.

The question then remains how to detect which types are sets and which are 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 a typeclass-like system such as given by Devriese and Piessens (2014).

A nicer, but probably also harder approach is to try to detect sets automatically. This problem is very hard in general, but we could at least try to detect easy cases like $\mathbb{N}$, using Hedberg's theorem or a generalization of it (Kraus et al. 2013). Hedberg's theorem states that if a type A has decidable equality, then it is a set. In particular, if D is a simple (non-indexed) data type such that each constructor is of the form $c : \Delta_c \rightarrow D \rightarrow \ldots \rightarrow D \rightarrow D$ where all of the types in Δ_c have decidable equality, then D itself also has decidable equality, hence it is a set by Hedberg's theorem. For example, this can be used to see that $\mathbb{N}$ is a set. 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.

5. 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), Idris (Brady 2013), and the Equations package for Coq (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

$$\begin{array}{l}\texttt{match}\ e\ \texttt{as}\ x\ \texttt{in}\ D\ \bar{u}\ \texttt{return}\ P\ \texttt{with}\\ \quad \mid c_1\ \bar{y}_1 \Rightarrow e_1\\ \quad \mid \ \ldots\\ \quad \mid c_n\ \bar{y}_n \Rightarrow e_n\\ \texttt{end}\end{array} \tag{55}$$

In the language of this paper, this corresponds to

$$\texttt{case}_D\ (\lambda \bar{u}; x.\ P)\ (\lambda \bar{y}_1.\ e_1)\ \ldots\ (\lambda \bar{y}_n.\ e_n)\ e \tag{56}$$

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

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.

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.

6. Conclusion and future work

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 away 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.

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 type checker of our languages perform the translation from pattern matching to a core calculus in practice. This is already done in the Equations package for Coq by Sozeau (2010), but they still need the K axiom for the translation. It would be interesting to see if our criterion could be integrated into this approach. Another very appealing idea is to write a compiler for dependent pattern matching inside the type theory by means of *datatype-generic programming* as described by Dagand (2013).

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 nontrivial 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 just limit the unification algorithm further. Our present paper gives a glimpse of how a theory of pattern matching with higher inductive types might look like, but future research will have to show how much of the original pattern matching algorithm can be salvaged.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven, and by the Research Foundation - Flanders under grant number G004321N. Jesper Cockx and Dominique Devriese both hold a Ph.D. fellowship of the Research Foundation - Flanders (FWO).

References

T. Altenkirch. Without-K problem, 2012. URL `https://lists.chalmers.se/pipermail/agda/2012/004104.html`. On the Agda mailing list.

E. Brady. Idris, a general purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5), 2013.

J. Cockx. Yet another way Agda –without-K is incompatible with univalence, 2014. URL `https://lists.chalmers.se/pipermail/agda/2014/006367.html`. On the Agda mailing list.

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

P.-E. Dagand. *A cosmology of datatypes: reusability and dependent types*. PhD thesis, University of Strathclyde, 2013.

N. A. Danielsson. Experiments related to equality, 2013. URL `http://www.cse.chalmers.se/~nad/repos/equality/`. Agda code.

D. Devriese and F. Piessens. Instance arguments in Agda. *Higher-order and symbolic computation*, 2014.

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

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

M. Hofmann and T. Streicher. The groupoid model refutes uniqueness of identity proofs. In *Logic in Computer Science*, pages 208–212, 1994.

J.-P. Jouannaud and C. Kirchner. *Solving equations in abstract algebras: A rule-based survey of unification*. 1990.

N. Kraus and C. Sattler. On the hierarchy of univalent universes: U(n) is not n-truncated. *arXiv preprint arXiv:1311.4002*, 2013.

N. Kraus, M. Escardó, T. Coquand, and T. Altenkirch. Generalizations of Hedberg's theorem. In *Typed Lambda Calculi and Applications*, pages 173–188. Springer, 2013.

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

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

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

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

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

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

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

U. Norell, A. Abel, and N. A. Danielsson. Release notes for Agda 2 version 2.3.2, 2012. URL `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Version-2-3-2`.

J. Reed. Another possible without-K problem, 2013. URL `https://lists.chalmers.se/pipermail/agda/2013/005578.html`. On the Agda mailing list.

A. Sicard-Ramírez. –without-K option too restrictive?, 2013. URL `https://lists.chalmers.se/pipermail/agda/2013/005407.html`. On the Agda mailing list.

M. Sozeau. Equations: A dependent pattern-matching compiler. In *Interactive theorem proving*, 2010.

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

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

A. A few homogeneous constructions on constructors

Case Analysis. $\mathtt{case_D}$ is given by dropping the inductive hypotheses from the eliminator, i.e. it is itself a $\bar{\mathtt{D}}$-elimination operator with methods

$$\begin{aligned} m_i : (\bar t : \Delta_i) \to \\ (x_1 : \Phi_{i1} \to D\ \bar v_{i1})\ \dots\ (x_{n_i} : \Phi_{in_i} \to D\ \bar v_{in_i}) \to \\ P\ \bar u_i\ (\mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i}) \end{aligned} \tag{57}$$

for $i = 1, \dots k$.

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

$$\begin{aligned} m_i = \lambda \bar t; x_1; \dots; x_{n_i}; h_1; \dots; h_{n_i}. \\ (\Phi_{i1} \to h_1\ \Phi_{i1} \times P\ \bar v_{i1}\ (x_1\ \Phi_{i1})) \times \dots \\ \times (\Phi_{in_i} \to h_{n_i}\ \Phi_{in_i} \times P\ \bar v_{in_i}\ (x_{n_i}\ \Phi_{in_i})) \end{aligned} \tag{58}$$

i.e. $\mathtt{Below_D}\ P\ x$ is a tuple asserting $P\ y$ for all y structurally smaller than x.

Next, 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:

$$\begin{aligned} m_i = \lambda \bar t; x_1; \dots; x_{n_i}; h_1; \dots; h_{n_i}. \\ (\lambda \Phi_{i1}.\ h_1\ \Phi_{i1},\ P\ \bar v_{i1}\ x_1\ (h_1\ \Phi_{i1})),\ \dots, \\ (\lambda \Phi_{in_i}.\ h_{n_i}\ \Phi_{in_i},\ P\ \bar v_{in_i}\ x_{n_i}\ (h_{n_i}\ \Phi_{in_i})) \end{aligned} \tag{59}$$

Finally, we define $\mathtt{rec_D}\ P\ p\ \bar{\mathtt{D}} := p\ \bar{\mathtt{D}}\ (\mathtt{below_D}\ P\ p\ \bar{\mathtt{D}})$.

No Confusion. First, we define $\mathtt{NoConfusion_D}\ \bar a\ \bar b$ by applying $\mathtt{case_D}$ 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_{ij} to fill in, one for each pair of constructors. On the diagonal (where $i = j$) we define $m_{ii} = \lambda \bar x; \bar x'.\ \bar x \equiv \bar x'$, and if $i \neq j$ we simply give $m_{ij} = \lambda \bar x; \bar x'.\ \bot$ (the empty type).

Next, we define $\mathtt{noConf_D}\ \bar a\ \bar b$. By telescopic substitution $\overline{\mathtt{subst}}$ with motive $\mathtt{NoConfusion_D}\ \bar a$, it is sufficient to give a function of type $(\bar a : \bar{\mathtt{D}}) \to \mathtt{NoConfusion_D}\ \bar a\ \bar a$. 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

$$\begin{aligned} m_{ij} : \mathtt{NoConfusion_D}\ (\bar u_i; \mathtt{c_i}\ \bar x)\ (\bar u'_j; \mathtt{c_j}\ \bar x') \to \\ \bar u_i\ (\mathtt{c_i}\ \bar x) \equiv \bar u'_j\ (\mathtt{c_j}\ \bar x') \end{aligned} \tag{60}$$

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 $\overline{\mathtt{refl}}$.

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

$$(\bar a\ \bar b : \bar{\mathtt{D}})(\bar e : \bar a \equiv \bar b) \to \mathtt{noConf_D}^{-1}\ \bar a\ \bar b\ (\mathtt{noConf_D}\ \bar a\ \bar b\ \bar e) \equiv \bar e \tag{61}$$

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

$$(\bar a : \bar{\mathtt{D}}) \to \mathtt{noConf_D}^{-1}\ \bar a\ \bar a\ (\mathtt{noConf_D}\ \bar a\ \bar a\ \overline{\mathtt{refl}}) \equiv \overline{\mathtt{refl}} \tag{62}$$

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

Acyclicity. 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 $\bar a \not\leq \bar b := \bar a \not< \bar b \times \bar a \not\equiv \bar b$. If $x : \mathtt{D}\ \bar u$ and $y : \mathtt{D}\ \bar v$ then we often write $x \not< y$ and $x \not\leq y$ instead of $\bar u; x \not< \bar v; y$ and $\bar u; x \not\leq \bar v; y$ to avoid too much clutter (of which we have enough already). Note that $x \not< \mathtt{c_i}\ \Delta_i\ x_1\ \dots\ x_{n_i} = (\Phi_{i1} \to x \not\leq x_1\ \Phi_{i1}) \times \dots \times (\Phi_{in_i} \to x \not\leq x_{n_i}\ \Phi_{in_i})$ by definition of $\mathtt{Below_D}$ and $\not\leq$.

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 $\mathtt{c_i} : \Delta_i \to (\Phi_{i1} \to \mathtt{D}\ \bar v_{i1}) \to \dots \to (\Phi_{in_i} \to \mathtt{D}\ \bar v_{in_i}) \to \mathtt{D}\ \bar u_i$ the subgoal $(\bar t : \Delta_i) \to (x_1 : \Phi_{i1} \to \mathtt{D}\ \bar v_{i1})\ \dots\ (x_{n_i} : \Phi_{in_i} \to \mathtt{D}\ \bar v_{in_i}) \to (h_1 : \Phi_{i1} \to x_1\ \Phi_{i1} \not< x_1\ \Phi_{i1})\ \dots\ (h_{n_i} : \Phi_{in_i} \to x_{n_i}\ \Phi_{in_i} \not< x_{n_i}\ \Phi_{in_i}) \to \mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i} \not< \mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i}$.

In order to continue, we first define the auxiliary types $\mathtt{Step_{ij}} : \Delta_i \to (x_1 : \Phi_{i1} \to \mathtt{D}\ \bar v_{i1})\ \dots\ (x_{n_i} : \Phi_{in_i} \to \mathtt{D}\ \bar v_{in_i}) \to \Phi_{ij} \to \bar{\mathtt{D}} \to \mathtt{Set}_d$ for $i = 1, \dots, k$ and $j = 1, \dots, n_i$ as follows:

$$\begin{aligned} \mathtt{Step_{ij}}\ \bar t\ x_1\ \dots\ x_{n_i}\ \Phi_{ij}\ (\bar u; b) = \\ (x_j\ \Phi_{ij}) \not< b \to (\mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i}) \not\leq b \end{aligned} \tag{63}$$

Now suppose that we can construct $\mathtt{step_{ij}} : (\bar t : \Delta_i) \to (x_1 : \Phi_{i1} \to \mathtt{D}\ \bar v_{i1})\ \dots\ (x_{n_i} : \Phi_{in_i} \to \mathtt{D}\ \bar v_{in_i}) \to \Phi_{ij} \to (\bar a : \bar{\mathtt{D}}) \to \mathtt{Step_{ij}}\ \bar t\ x_1\ \dots\ x_{n_i}\ \Phi_{ij}\ \bar a$. Then we can solve the subgoal by filling in

$$\begin{aligned} \lambda \bar t; \bar x; \bar h. \\ (\lambda \Phi_{i1}.\ \mathtt{step_{i1}}\ \bar t\ \bar x\ \Phi_{i1}\ \bar v_{i1}\ (x_1\ \Phi_{i1})\ (h_1\ \Phi_{i1})), \\ \dots, \\ (\lambda \Phi_{in_i}.\ \mathtt{step_{in_i}}\ \bar t\ \bar x\ \Phi_{in_i}\ \bar v_{in_i}\ (x_{n_i}\ \Phi_{in_i})\ (h_{n_i}\ \Phi_{in_i})) \end{aligned} \tag{64}$$

So we only need to construct the $\mathtt{step_{ij}}$.

The construction of $\mathtt{step_{ij}}\ \bar t\ x_1\ \dots\ x_{n_i}\ \Phi_{ij} : (\bar a : \bar{\mathtt{D}}) \to \mathtt{Step_{ij}}\ \bar t\ x_1\ \dots\ x_{n_i}\ \Phi_{ij}\ \bar a$ proceeds by applying $\mathtt{elim_D}$ with motive $\mathtt{Step_{ij}}\ \bar t\ x_1\ \dots\ x_{n_i}\ \Phi_{ij}$. The new subgoals are of the form

$$\begin{aligned} (\bar t' : \Delta'_p)(x'_1 : \Phi'_{p1} \to \mathtt{D}\ \bar v'_{p1})\ \dots\ (x'_{n_p} : \Phi'_{pn_p} \to \mathtt{D}\ \bar v'_{pn_p}) \to \\ (h_1 : (\bar s'_1 : \Phi'_{p1}) \to \mathtt{Step_{ij}}\ \bar t\ \bar x\ \Phi_{ij}\ \bar v'_{p1}\ (x'_1\ \bar s'_1))\ \dots \\ (h_{n_p} : (\bar s'_{n_p} : \Phi'_{pn_p}) \to \mathtt{Step_{ij}}\ \bar t\ \bar x\ \Phi_{ij}\ \bar v'_{pn_p}\ (x'_{n_p}\ \bar s'_{n_p})) \to \\ \mathtt{Step_{ij}}\ \bar t\ \bar x\ \Phi_{ij}\ \bar u'_p\ (\mathtt{c_p}\ \bar t'\ \bar x') \end{aligned} \tag{65}$$

We solve them by giving:

$$\lambda \bar t'; x'_1; \dots; x'_{n_p}; h_1; \dots; h_{n_p}; H.\ \alpha, \beta \tag{66}$$

where we still have to construct

$$\alpha : \mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i} \not< \mathtt{c_p}\ \bar t'\ x'_1\ \dots\ x'_{n_p} \tag{67}$$

and

$$\beta : \mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i} \not\equiv \mathtt{c_p}\ \bar t'\ x'_1\ \dots\ x'_{n_p} \tag{68}$$

For any $\bar s : \Phi_{ij}$, we have $H : x_j\ \bar s \not< \mathtt{c_p}\ \Delta'_p\ x'_1\ \dots\ x'_{n_p}$ or, by definition of $\not<$, $H = (H_1, \dots, H_{n_p})$ where $H_q : (\bar s' : \Phi'_{pq}) \to x_j\ \bar s \not\leq x'_q\ \bar s'$.

The construction of α reduces to the construction of components $\alpha_q : \Phi'_{pq} \to \mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i} \not\leq x'_q\ \Phi'_{pq}$. But these we can give as $\alpha_q = \lambda \bar s'.\ h_q\ (\pi_1\ (H_p\ \bar s'))$ (where π_1 is projection onto the first component).

For constructing β, we assume $\mathtt{c_i}\ \bar t\ x_1\ \dots\ x_{n_i} \equiv \mathtt{c_p}\ \bar t'\ x'_1\ \dots\ x'_{n_p}$ and derive an element of $\bot$. By $\mathtt{noConf_D}$, it suffices to consider the case where $i = p$, $\Delta_i = \Delta'_i$, and $x_1; \dots; x_{n_i} = x'_1, \dots, x'_{n_i}$. But then we have $H_j\ \bar s : x_j\ \bar s \not\leq x_j\ \bar s$, hence $\pi_2\ (H_j\ \Phi_{ij})\ \overline{\mathtt{refl}} : \bot$. This finishes the construction of $\mathtt{noCycle_D}$.