Dependent pattern matching and proof-relevant unification

Jesper Cockx

Gothenburg University / Chalmers

19 February 2017

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

data \mathbb{N} : Type where

zero: N

 $\verb"suc":\mathbb{N}\to\mathbb{N}$

```
data \mathbb{N}: Type where
   zero: N
   \operatorname{suc}:\mathbb{N}\to\mathbb{N}
minimum: \mathbb{N} \to \mathbb{N} \to \mathbb{N}
minimum zero y = zero
minimum (suc x) zero = { }
minimum (suc x) (suc y) = { }
```

```
data \mathbb{N}: Type where
   zero: N
   \operatorname{suc}:\mathbb{N}\to\mathbb{N}
minimum: \mathbb{N} \to \mathbb{N} \to \mathbb{N}
minimum zero y
                           = zero
minimum (suc x) zero = zero
minimum (\operatorname{suc} x) (\operatorname{suc} y) = { }
```

```
data \mathbb{N}: Type where
   zero: N
   \operatorname{suc}:\mathbb{N}\to\mathbb{N}
minimum: \mathbb{N} \to \mathbb{N} \to \mathbb{N}
minimum zero y
                         = zero
minimum (suc x) zero = zero
minimum (suc x) (suc y) = suc (minimum x y)
```

Why pattern matching?

- Intuitive way to write definitions by case analysis and recursion / induction.
- Computational meaning is obvious.
- Automates boring equational reasoning.

```
data Vec(A: Type): \mathbb{N} \to Type where

nil: Vec A zero

cons: (n: \mathbb{N}) \to A \to Vec A n \to Vec A (suc n)
```

```
data Vec (A: Type): \mathbb{N} \to \text{Type} where
   nil : Vec A zero
   cons : (n: \mathbb{N}) \to A \to \text{Vec } A \ n \to \text{Vec } A \ (\text{suc } n)

tail: (k: \mathbb{N}) \to \text{Vec } A \ (\text{suc } k) \to \text{Vec } A \ k

tail k xs = {}
```

```
data Vec (A: Type): \mathbb{N} \to \text{Type} where
  nil : Vec A zero
  cons : (n: \mathbb{N}) \to A \to \text{Vec } A \ n \to \text{Vec } A \ (\text{suc } n)

tail: (k: \mathbb{N}) \to \text{Vec } A \ (\text{suc } k) \to \text{Vec } A \ k

tail k nil = { } -- suc k = zero

tail k (cons n x xs) = { } -- suc k = suc n
```

```
data Vec (A: Type): \mathbb{N} \to \text{Type} where
    nil : Vec A zero
    cons : (n: \mathbb{N}) \to A \to Vec A n \to \text{Vec } A (suc n)

tail: (k: \mathbb{N}) \to Vec A (suc k) \to Vec A k

tail k nil = { } -- impossible

tail k (cons n \times xs) = { } -- suc k = suc n
```

```
data Vec (A: Type): \mathbb{N} \to \text{Type} where
   nil : Vec A zero
   cons : (n: \mathbb{N}) \to A \to \text{Vec } A \ n \to \text{Vec } A \ (\text{suc } n)

tail: (k: \mathbb{N}) \to \text{Vec } A \ (\text{suc } k) \to \text{Vec } A \ k

tail k (cons n x xs) = {} -- suc k = suc n
```

```
data Vec (A: Type): \mathbb{N} \to \text{Type} where
   nil : Vec A zero
   cons : (n: \mathbb{N}) \to A \to Vec A n \to \text{Vec } A (suc n)

tail : (k: \mathbb{N}) \to \text{Vec } A (suc k) \to \text{Vec } A k

tail .n (cons n \times xs) = {}
```

```
data Vec (A : Type) : \mathbb{N} \to Type where

nil : Vec A zero

cons : (n : \mathbb{N}) \to A \to Vec A n \to Vec A (suc n)

tail : (k : \mathbb{N}) \to Vec A (suc k) \to Vec A k

tail .n (cons n x xs) = xs
```

```
data \_ \equiv \_(x : A) : A \rightarrow \text{Type where}
refl: x \equiv x

trans: (x \ y \ z : A) \rightarrow (x \equiv y) \rightarrow (y \equiv z) \rightarrow (x \equiv z)
trans x \ y \ z \ p \ q = \{\}
```

```
data \_ \equiv \_(x : A) : A \to \text{Type where}

refl: x \equiv x

trans: (x \ y \ z : A) \to (x \equiv y) \to (y \equiv z) \to (x \equiv z)

trans x . x z \text{ refl } q = \{\}
```

```
data \_ \equiv \_(x : A) : A \to \text{Type where}
refl: x \equiv x

trans: (x \ y \ z : A) \to (x \equiv y) \to (y \equiv z) \to (x \equiv z)
trans x . x . x \text{ refl refl} = \{\}
```

```
data \_ \equiv \_(x : A) : A \rightarrow \text{Type where}
refl: x \equiv x

trans: (x \ y \ z : A) \rightarrow (x \equiv y) \rightarrow (y \equiv z) \rightarrow (x \equiv z)
trans x \ .x \ .x \ refl \ refl = refl
```

Pattern matching implies uniqueness of identity proofs

```
\begin{array}{ll} \text{UIP}: (x : A)(e : x \equiv_A x) \rightarrow e \equiv_{x \equiv_{A^X}} \text{refl} \\ \text{UIP} \ x \ e &= \left\{ \begin{array}{c} \\ \end{array} \right\} \end{array}
```

Pattern matching implies uniqueness of identity proofs

Pattern matching implies uniqueness of identity proofs

```
UIP: (x : A)(e : x \equiv_A x) \rightarrow e \equiv_{x \equiv_A x} refl
UIP x refl = refl
```

UIP is incompatible with univalence

UIP is incompatible with univalence:

 UIP implies coerce e true = true for all e : Bool ≡ Bool

UIP is incompatible with univalence:

- UIP implies coerce e true = true for all e : Bool ≡ Bool
- Univalence gives swap : Bool ≡ Bool such that coerce swap true = false

UIP is incompatible with univalence:

- UIP implies coerce e true = true for all e : Bool ≡ Bool
- Univalence gives swap : Bool ≡ Bool such that coerce swap true = false

hence true = false!

Pattern matching without K

How to avoid proofs of UIP in general?

Pattern matching without K

How to avoid proofs of UIP in general?

Answer: by disabling the deletion rule of the unification algorithm.

Pattern matching without K

How to avoid proofs of UIP in general?

Answer: by disabling the deletion rule of the unification algorithm.

This calls for a deeper investigation of unification in dependent type theory!

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

Specialization by unification

We use unification to . . .

- eliminate impossible cases
- specialize the result type

Specialization by unification

We use unification to . . .

- eliminate impossible cases
- specialize the result type

The output of unification can change our notion of equality!

Specialization by unification

We use unification to . . .

- eliminate impossible cases
- specialize the result type

The output of unification can change our notion of equality!

Main question: How to make sure the output of unification is correct?

Proof-relevant unification

We want a unification algorithm that

- ... takes types of equations into account.
- ... represents input and output internally.
- ... produces *evidence* that its output is correct.

A unification problem consists of

- 1. Flexible variables $x_1 : A_1, x_2 : A_2, \ldots$
- 2. Equations $u_1 = v_1 : B_1, ...$

A unification problem consists of

- 1. Flexible variables $x_1 : A_1, x_2 : A_2, \dots$
- 2. Equations $u_1 = v_1 : B_1, ...$

This can be represented as a **telescope**:

$$(x_1 : A_1)(x_2 : A_2) \dots$$

 $(e_1 : u_1 \equiv_{B_1} v_1)(e_2 : u_2 \equiv_{B_2} v_2) \dots$

e.g.
$$(k : \mathbb{N})(n : \mathbb{N})(e : \operatorname{suc} k \equiv_{\mathbb{N}} \operatorname{suc} n)$$

A unification problem consists of

- Flexible variables Γ
- 2. Equations $u_1 = v_1 : B_1, ...$

This can be represented as a **telescope**:

```
 (e_1:u_1\equiv_{B_1}v_1)(e_2:u_2\equiv_{B_2}v_2)\dots  e.g. (k:\mathbb{N})(n:\mathbb{N})(e:\operatorname{\mathtt{suc}} k\equiv_{\mathbb{N}}\operatorname{\mathtt{suc}} n)
```

A unification problem consists of

- Flexible variables Γ
- 2. Equations $\bar{u} = \bar{v} : \Delta$

This can be represented as a **telescope**:

$$\Gamma(\bar{e}:\bar{u}\equiv_{\Delta}\bar{v})$$

e.g.
$$(k : \mathbb{N})(n : \mathbb{N})(e : \operatorname{suc} k \equiv_{\mathbb{N}} \operatorname{suc} n)$$

Unifiers as telescope maps

A unifier of \bar{u} and \bar{v} is a substitution $\sigma: \Gamma' \to \Gamma$ such that $\bar{u}\sigma = \bar{v}\sigma$.

Unifiers as telescope maps

A unifier of \bar{u} and \bar{v} is a substitution $\sigma: \Gamma' \to \Gamma$ such that $\bar{u}\sigma = \bar{v}\sigma$.

This can be represented as a telescope map:

$$f: \Gamma' \to \Gamma(\bar{e}: \bar{u} \equiv_{\Delta} \bar{v})$$

e.g.
$$f:() \rightarrow (n:\mathbb{N})(e:n \equiv_{\mathbb{N}} \mathsf{zero})$$

A map $f:() \to (n:\mathbb{N})(e:n \equiv_{\mathbb{N}} \mathbf{zero})$ gives us two things:

A map $f:() \to (n:\mathbb{N})(e:n \equiv_{\mathbb{N}} \mathbf{zero})$ gives us two things:

1. A **value** for *n* such that $n \equiv_{\mathbb{N}} \mathbf{zero}$

A map $f:() \to (n:\mathbb{N})(e:n \equiv_{\mathbb{N}} \mathtt{zero})$ gives us two things:

- 1. A **value** for *n* such that $n \equiv_{\mathbb{N}} \mathbf{zero}$
- 2. Explicit **evidence** e of $n \equiv_{\mathbb{N}} \mathbf{zero}$

A map $f:() \to (n:\mathbb{N})(e:n \equiv_{\mathbb{N}} \mathtt{zero})$ gives us two things:

- 1. A **value** for *n* such that $n \equiv_{\mathbb{N}} \mathbf{zero}$
- 2. Explicit **evidence** e of $n \equiv_{\mathbb{N}} \mathbf{zero}$

 \implies Unification is guaranteed to respect \equiv !

Three valid unifiers

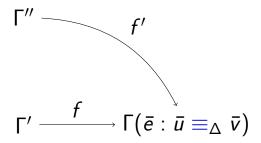
```
f_1:(k:\mathbb{N})\to (k\ n:\mathbb{N})(e:k\equiv_\mathbb{N} n)
f_1 k = k; k; refl
f_2:() \to (k \ n: \mathbb{N})(e: k \equiv_{\mathbb{N}} n)
f_2() = zero; zero; refl
f_3:(k n:\mathbb{N})\to (k n:\mathbb{N})(e:k\equiv_{\mathbb{N}} n)
f_3 k n = k : k : refl
```

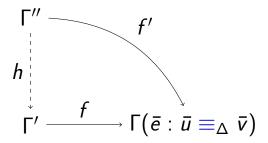
A most general unifier of \bar{u} and \bar{v} is a unifier σ such that for any σ' with $\bar{u}\sigma' = \bar{v}\sigma'$, there is a ν such that $\sigma' = \sigma \circ \nu$.

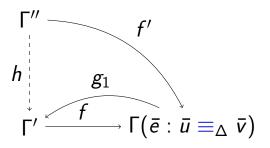
A most general unifier of \bar{u} and \bar{v} is a unifier σ such that for any σ' with $\bar{u}\sigma' = \bar{v}\sigma'$, there is a ν such that $\sigma' = \sigma \circ \nu$.

To avoid 'ghost variables', ν should be unique.

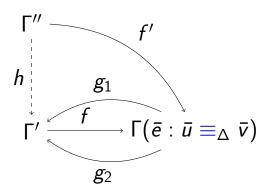
$$\Gamma' \xrightarrow{f} \Gamma(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v})$$



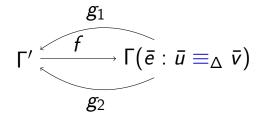




 $(\forall f': h \text{ exists}) \Leftrightarrow f \text{ has a } right \text{ inverse } g_1$



 $(\forall f' : h \text{ exists}) \Leftrightarrow f \text{ has a } right \text{ inverse } g_1$ $(\forall f' : h \text{ is unique}) \Leftrightarrow f \text{ has a } left \text{ inverse } g_2$



f has a right inverse g_1 f has a left inverse g_2

Most general unifiers are equivalences!

$$f:\Gamma(\bar{e}:\bar{u}\equiv_{\Delta}\bar{v})\simeq\Gamma'$$

 $(k n : \mathbb{N})(e : \operatorname{suc} k \equiv_{\mathbb{N}} \operatorname{suc} n)$

```
(k \ n : \mathbb{N})(e : \operatorname{suc} k \equiv_{\mathbb{N}} \operatorname{suc} n)
(k \ n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)
```

```
(k \ n : \mathbb{N})(e : \mathbf{suc} \ k \equiv_{\mathbb{N}} \mathbf{suc} \ n)
(k \ n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)
(k \ n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)
(k : \mathbb{N})
```

```
(k n : \mathbb{N})(e : \mathbf{suc} \ k \equiv_{\mathbb{N}} \mathbf{suc} \ n)
                         (k n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)
                                         (k:\mathbb{N})
f:(k:\mathbb{N})\to (k\;n:\mathbb{N})(e:\mathit{suc}\;k\equiv_{\mathbb{N}}\mathit{suc}\;n)
```

f k = k: k: refl

The solution rule

solution:
$$(x : A)(e : x \equiv_A t) \simeq ()$$

The deletion rule

deletion:
$$(e: t \equiv_A t) \simeq ()$$

The injectivity rule

Negative unification rules

A negative unification rule applies to impossible equations, e.g. suc x = zero.

Negative unification rules

A negative unification rule applies to impossible equations, e.g. suc x = zero.

This can be represented by an equivalence:

$$(e: \mathtt{suc}\ x \equiv_{\mathbb{N}} \mathtt{zero}) \simeq \bot$$

where \perp is the **empty type**.

The conflict rule

```
\mathsf{conflict}_{\mathtt{suc},\mathtt{zero}}: (e:\mathtt{suc}\ x\equiv_{\mathbb{N}}\mathtt{zero})\simeq \bot
```

The cycle rule

$$\operatorname{cycle}_{n,\operatorname{\mathtt{suc}} n}:(e:n\equiv_{\mathbb{N}}\operatorname{\mathtt{suc}} n)\simeq\bot$$

Heterogeneous equations

Problem: What is the type of e_2 ?

```
(e:(0,\mathtt{nil})\equiv_{\Sigma_{n:\mathbb{N}}} 	ext{Vec } A_n\ (1,\mathtt{cons}\ 0\ x\ xs))
|c|
(e_1:0\equiv_{\mathbb{N}}1)(e_2:\mathtt{nil}\equiv_{\mathtt{Vec}\ A\,???}\mathtt{cons}\ 0\ x\ xs)
```

Heterogeneous equations

Problem: What is the type of e_2 ?

Solution: keep track of dependencies by

introducing a new variable for each equation

```
(e:(0,\mathtt{nil})\equiv_{\Sigma_{n:\mathbb{N}}} Vec _{An}(1,\mathtt{cons}\ 0\ x\ xs))
|_{\mathcal{C}}
(e_1:0\equiv_{\mathbb{N}}1)(e_2:\mathtt{nil}\equiv_{\mathtt{Vec}\ A\ e_1}\ \mathtt{cons}\ 0\ x\ xs)
```

Heterogeneous equations

Problem: What is the type of e_2 ?

Solution: keep track of dependencies by

introducing a new variable for each equation

$$(e:(0,\mathtt{nil})\equiv_{\Sigma_{n:\mathbb{N}}\mathsf{Vec}\;A\;n}(1,\mathtt{cons}\;0\;x\;xs))$$
 $(e_1:0\equiv_{\mathbb{N}}1)(e_2:\mathtt{nil}\equiv_{\mathtt{Vec}\;A\;e_1}\mathtt{cons}\;0\;x\;xs)$

This is called a *telescopic equality*.

```
(e:(\mathsf{Bool},\mathsf{true})\equiv_{\Sigma_{A:\mathsf{Type}}A}(\mathsf{Bool},\mathsf{false}))
```

```
(e:(\mathsf{Bool},\mathsf{true})\equiv_{\Sigma_{A:\mathrm{Type}}A}(\mathsf{Bool},\mathsf{false}))
|c|
(e_1:\mathsf{Bool}\equiv_{\mathrm{Type}}\mathsf{Bool})(e_2:\mathsf{true}\equiv_{e_1}\mathsf{false})
```

```
(e:(\mathsf{Bool},\mathsf{true})\equiv_{\Sigma_{A:\mathsf{Type}}A}(\mathsf{Bool},\mathsf{false}))
(e_1:\mathsf{Bool}\equiv_{\mathsf{Type}}\mathsf{Bool})(e_2:\mathsf{true}\equiv_{e_1}\mathsf{false})
\downarrow \wr
```

```
(e:(\mathsf{Bool},\mathsf{true}) \equiv_{\Sigma_{A:\mathsf{Type}}\mathcal{A}} (\mathsf{Bool},\mathsf{false}))
(e_1:\mathsf{Bool} \equiv_{\mathsf{Type}} \mathsf{Bool})(e_2:\mathsf{true} \equiv_{e_1} \mathsf{false})
```

The conflict rule does not apply!

```
(e:(Bool,true)\equiv_{\Sigma_{A:Type}Bool}(Bool,false))
```

```
(e:(\mathsf{Bool},\mathsf{true})\equiv_{\Sigma_{A:\mathsf{Type}}\mathsf{Bool}}(\mathsf{Bool},\mathsf{false}))
|c|
(e_1:\mathsf{Bool}\equiv_{\mathsf{Type}}\mathsf{Bool})(e_2:\mathsf{true}\equiv_{\mathsf{Bool}}\mathsf{false})
```

```
(e:(\mathsf{Bool},\mathsf{true}) \equiv_{\Sigma_{A:\mathsf{Type}}\mathsf{Bool}} (\mathsf{Bool},\mathsf{false}))
|c|
(e_1:\mathsf{Bool} \equiv_{\mathsf{Type}} \mathsf{Bool})(e_2:\mathsf{true} \equiv_{\mathsf{Bool}} \mathsf{false})
|c|
```

Whether a unification rule can be applied depends on the **type** of the equation!

Injectivity for indexed data

Idea: simplify equations between indices together with equation between constructors:

```
(e_1: \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ n)
(e_2: \mathtt{cons}\ k \ x \ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n \ y \ ys)
```

Injectivity for indexed data

Idea: simplify equations between indices together with equation between constructors:

```
(e_1: \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ n)
(e_2: \mathtt{cons}\ k\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys)
(e_1': k \equiv_{\mathbb{N}} n)(e_2': x \equiv_{A} y)
(e_3': xs \equiv_{\mathtt{Vec}\ A\ e_1} ys)
```

Injectivity for indexed data

Idea: simplify equations between indices together with equation between constructors:

```
(e_1: \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ n)
(e_2: \mathtt{cons}\ k\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys)
(e_1': k \equiv_{\mathbb{N}} n)(e_2': x \equiv_{A} y)
(e_3': xs \equiv_{\mathtt{Vec}\ A\ e_1} ys)
```

Length of the Vec must be *fully general*: must be an equation variable.

data Im $(f : A \rightarrow B) : B \rightarrow \text{Type}$ where image : $(x : A) \rightarrow \text{Im } f (f x)$

```
data \operatorname{Im} (f : A \to B) : B \to \operatorname{Type} where \operatorname{image} : (x : A) \to \operatorname{Im} f (f x)
```

```
(x_1 \ x_2 : A)(e_1 : f \ x_1 \equiv_B f \ x_2)
(e_2 : image \ x_1 \equiv_{Im \ f \ e_1} image \ x_2)
```

```
data \operatorname{Im} (f : A \to B) : B \to \operatorname{Type} where \operatorname{image} : (x : A) \to \operatorname{Im} f (f x)
```

```
(x_1 \ x_2 : A)(e_1 : f \ x_1 \equiv_B f \ x_2)

(e_2 : image \ x_1 \equiv_{Im \ f \ e_1} image \ x_2)

| ? 

(x_1 \ x_2 : A)(e : x_1 \equiv_A x_2)
```

```
data Im (f : A \rightarrow B) : B \rightarrow \text{Type} where image : (x : A) \rightarrow \text{Im } f (f x)
```

```
(x_1 \ x_2 : A)(e_1 : f \ x_1 \equiv_B f \ x_2)

(e_2 : image \ x_1 \equiv_{Im \ f \ e_1} image \ x_2)

(x_1 \ x_2 : A)(e : x_1 \equiv_A x_2)

(x_1 : A)
```

Dependent pattern matching

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

```
(e: cons \ n \ x \ xs \equiv_{Vec \ A \ (suc \ n)} cons \ n \ y \ ys)
```

```
(e: cons \ n \ x \ xs \equiv_{\text{Vec } A \ (suc \ n)} cons \ n \ y \ ys)
(e_1: suc \ n \equiv_{\mathbb{N}} suc \ n)
(e_2: cons \ n \ x \ xs \equiv_{\text{Vec } A \ e_1} cons \ n \ y \ ys)
(p: e_1 \equiv_{\text{suc } n \equiv_{\mathbb{N}} suc \ n} refl)
```

```
(e: cons \ n \ x \ xs \equiv_{Vec \ A \ (suc \ n)} cons \ n \ y \ ys)
                         (e_1 : suc \ n \equiv_{\mathbb{N}} suc \ n)
      (e_2 : cons \ n \ x \ xs \equiv_{Vec \ A \ e_1} cons \ n \ y \ ys)
                    (p:e_1\equiv_{\mathtt{suc}\ n\equiv_{\mathbb{N}}\mathtt{suc}\ n}\mathtt{refl})
(e'_1: n \equiv_{\mathbb{N}} n)(e'_2: x \equiv_{A} y)(e'_3: xs \equiv_{\text{Vec } A e'_1} ys)
         (p: \texttt{cong suc } e_1' \equiv_{\texttt{suc } n \equiv_{\mathbb{N}} \texttt{suc } n} \texttt{refl})
```

```
(e: cons \ n \ x \ xs \equiv_{Vec \ A \ (suc \ n)} cons \ n \ y \ ys)
                          (e_1 : \mathsf{suc} \ n \equiv_{\mathbb{N}} \mathsf{suc} \ n)
       (e_2 : cons \ n \ x \ xs \equiv_{Vec \ A \ e_1} cons \ n \ y \ ys)
                     (p:e_1\equiv_{\mathtt{suc}\ n\equiv_{\mathbb{N}}\mathtt{suc}\ n}\mathtt{refl})
(e'_1:n\equiv_{\mathbb{N}} n)(e'_2:x\equiv_{A} y)(e'_3:xs\equiv_{\mathtt{Vec}\ A\ e'_1} ys)
          (p: \texttt{cong suc } e_1' \equiv_{\texttt{suc } n \equiv_{\mathbb{N}} \texttt{suc } n} \texttt{refl})
```

Higher-dimensional equations

$$(e'_1: n \equiv_{\mathbb{N}} n)(e'_2: x \equiv_{A} y)(e'_3: xs \equiv_{\text{Vec } A e'_1} ys)$$
$$(p: \text{cong suc } e'_1 \equiv_{\text{suc } n \equiv_{\mathbb{N}} \text{suc } n} \text{refl})$$

We call an equation between equality proofs (e.g. p) a **higher-dimensional equation**.

How to solve higher-dimensional equations?

Existing unification rules do not apply...

How to solve higher-dimensional equations?

Existing unification rules do not apply...

We solve the problem in three steps:

- 1. lower the dimension of equations
- 2. solve lower-dimensional equations
- 3. lift unifier to higher dimension

Step 1: lower the dimension of equations

We replace all equation variables by regular variables: instead of

$$(e_1: n \equiv_{\mathbb{N}} n)(e_2: x \equiv_{A} y)(e_3: xs \equiv_{\text{Vec } A e_1} ys)$$
$$(p: \text{cong suc } e_1 \equiv_{\text{suc } n \equiv_{\mathbb{N}} \text{suc } n} \text{refl})$$

let's first consider

```
(k:\mathbb{N})(u:A)(us:\operatorname{Vec} A\ k)(e:\operatorname{suc} k\equiv_{\mathbb{N}}\operatorname{suc} n)
```

Step 2: solve lower-dimensional equations

This gives us an equivalence f of type

```
(k : \mathbb{N})(u : A)(us : \operatorname{Vec} A k)

(e : \operatorname{suc} k \equiv_{\mathbb{N}} \operatorname{suc} n)

(u : A)(us : \operatorname{Vec} A n)
```

Step 3: lift unifier to higher dimension

We lift f to an equivalence f^{\uparrow} of type

```
(e_1: n \equiv_{\mathbb{N}} n)(e_2: x \equiv_A y)
(e_3: xs \equiv_{\operatorname{Vec} A e_1} ys)
(p: \operatorname{cong} \operatorname{suc} e_1 \equiv_{\operatorname{suc} n \equiv_{\mathbb{N}} \operatorname{suc} n} \operatorname{refl})
(e_2: x \equiv_A y)(e_3: xs \equiv_{\operatorname{Vec} A n} ys)
```

Final result of steps 1-3

```
(e : \mathbf{cons} \ n \ x \ xs \equiv_{\mathbf{Vec} \ A \ (\mathbf{suc} \ n)} \mathbf{cons} \ n \ y \ ys)
|c|
(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\mathbf{Vec} \ A \ n} ys)
```

Final result of steps 1-3

```
(e : \operatorname{cons} n \times xs \equiv_{\operatorname{Vec} A \operatorname{(suc} n)} \operatorname{cons} n y ys)
(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\operatorname{Vec} A n} ys)
```

This is the **forcing rule** for **cons**.

Lifting equivalences: (mostly) general case

Theorem. If we have an equivalence f of type

$$(x:A)(e:b_1 x \equiv_{B \times} b_2 x) \simeq C$$

we can construct f^{\uparrow} of type

$$(e: u \equiv_{A} v)(p: \operatorname{cong} b_{1} e \equiv_{r \equiv_{B} e^{S}} \operatorname{cong} b_{2} e)$$

$$(e': f u r \equiv_{C} f v s)$$

Dependent pattern matching

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

1. Translate the definition to a case tree.

- 1. Translate the definition to a case tree.
- 2. Translate each case split, using basic case_D-analysis.

- 1. Translate the definition to a case tree.
- 2. Translate each case split, using basic case_D-analysis.
- 3. Solve equations in each subtree, using specialization by unification.

- 1. Translate the definition to a case tree.
- 2. Translate each case split, using basic case_D-analysis.
- 3. Solve equations in each subtree, using specialization by unification.
- 4. Fill in recursive calls, using well-founded induction.

From pattern matching . . .

...to a case tree ...

```
 x \ y \ \underline{u} \ v \left\{ \begin{array}{l} . {\tt zero} \ y \ ({\tt lz} \ . y) \ \underline{u} \\ \big\{ . {\tt zero} \ . {\tt zero} \ ({\tt lz} \ . {\tt zero}) \ ({\tt lz} \ . {\tt zero}) \mapsto {\tt refl} \\ . ({\tt suc} \ k) \ . ({\tt suc} \ l) \ ({\tt ls} \ k \ l \ u) \ \underline{v} \\ \big\{ . ({\tt suc} \ k) \ . ({\tt suc} \ l) \ ({\tt ls} \ k \ l \ u) \ ({\tt ls} \ . l \ . k \ v) \\ \big\} \mapsto {\tt cong} \ {\tt suc} \ ({\tt antisym} \ k \ l \ u \ v) \end{array} \right.
```

... to eliminators.

```
antisym: (x y : \mathbb{N}) \to x < y \to y < x \to x \equiv_{\mathbb{N}} y
antisym = elim<(\lambda x y u. y < x \rightarrow x \equiv_{\mathbb{N}} y)
    (\lambda / v. elim_{<} (\lambda y \times v. \times \equiv_{\mathbb{N}} zero \rightarrow x \equiv_{\mathbb{N}} y)
        (\lambda x e. e)
        (\lambda l \, k_{--} e. \, elim_{\perp} \, (suc \, k \equiv_{\mathbb{N}} suc \, l) \, (noConf_{\mathbb{N}} \, (suc \, k) \, zero \, e))
         /zero v refl)
    (\lambda k I_- H v. \text{ cong suc } (H))
        (\text{elim}_{<} (\lambda x y_{-}. x \equiv_{\mathbb{N}} \text{suc } I \rightarrow u \equiv_{\mathbb{N}} \text{suc } k \rightarrow I < k)
             (\lambda l' e_{-}.elim_{+} (l < k) (noConf_{N} zero (suc l) e))
             (\lambda k' l' v'_{-} e_1 e_2. subst (\lambda n. n < k)
                 (noConf_N (suc k') (suc l) e_1)
                 (subst (\lambda m. \ k' < m) (noConf<sub>N</sub> (suc l') (suc k) e_2) v'))
             (\operatorname{suc} I)(\operatorname{suc} k) v \operatorname{refl} \operatorname{refl}))
```