Dependent pattern matching and proof-relevant unification

Jesper Cockx

Gothenburg University / Chalmers

19 February 2017

Dependent pattern matching

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

Dependent pattern matching

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

Simple pattern matching

$$\begin{array}{l}
\textbf{data}\ \mathbb{N} : \mathtt{Type}\ \textbf{where} \\
\quad \mathtt{zero} : \mathbb{N} \\
\quad \mathtt{suc}\ \ : \mathbb{N} \to \mathbb{N}
\end{array}$$

Simple pattern matching

```
data ℕ : Type where
  zero : ℕ
  suc  : ℕ → ℕ

minimum : ℕ → ℕ → ℕ
minimum x     y     = { }
```

Simple pattern matching

$$
\begin{array}{l}
\textbf{data } \mathbb{N} : \texttt{Type} \textbf{ where} \\
\quad \texttt{zero} : \mathbb{N} \\
\quad \texttt{suc} \;\; : \mathbb{N} \to \mathbb{N}
\end{array}
$$

$$
\begin{array}{llll}
\texttt{minimum} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} & & & \\
\texttt{minimum}\ \texttt{zero} & y & = & \{\ \ \} \\
\texttt{minimum}\ (\texttt{suc}\ x) & y & = & \{\ \ \}
\end{array}
$$

Simple pattern matching

$\mathbf{data}\ \mathbb{N} : \mathtt{Type}\ \mathbf{where}$
$\quad \mathtt{zero} : \mathbb{N}$
$\quad \mathtt{suc} \ \ : \mathbb{N} \to \mathbb{N}$

$$
\begin{aligned}
&\mathtt{minimum} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\\
&\mathtt{minimum}\ \mathtt{zero}\ \ \ \ y = \mathtt{zero}\\
&\mathtt{minimum}\ (\mathtt{suc}\ x)\ y = \boxed{\{\ \}}
\end{aligned}
$$

Simple pattern matching

data $\mathbb{N}$: Type **where**
  zero : $\mathbb{N}$
  suc : $\mathbb{N} \to \mathbb{N}$

```
minimum : ℕ → ℕ → ℕ
minimum zero    y       = zero
minimum (suc x) zero    = { }
minimum (suc x) (suc y) = { }
```

Simple pattern matching

data $\mathbb{N}$: Type **where**

```
  zero : ℕ
  suc  : ℕ → ℕ
```

$\texttt{minimum} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$

$\texttt{minimum zero} \quad y \qquad = \texttt{zero}$

$\texttt{minimum} \; (\texttt{suc} \; x) \; \texttt{zero} \qquad = \texttt{zero}$

$\texttt{minimum} \; (\texttt{suc} \; x) \; (\texttt{suc} \; y) \quad = \{ \ \}$

Simple pattern matching

data $\mathbb{N}$: Type **where**
  zero : $\mathbb{N}$
  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.

Dependent pattern matching

$$
\begin{array}{l}
\textbf{data}\ \texttt{Vec}\ (A : \texttt{Type}) : \mathbb{N} \to \texttt{Type}\ \textbf{where} \\
\quad \texttt{nil} \quad : \texttt{Vec}\ A\ \texttt{zero} \\
\quad \texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)
\end{array}
$$

Dependent pattern matching

$$
\begin{aligned}
&\textbf{data}\ \texttt{Vec}\ (A : \texttt{Type}) : \mathbb{N} \to \texttt{Type}\ \textbf{where}\\
&\quad \texttt{nil}\ \ : \texttt{Vec}\ A\ \texttt{zero}\\
&\quad \texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{tail} : (k : \mathbb{N}) \to \texttt{Vec}\ A\ (\texttt{suc}\ k) \to \texttt{Vec}\ A\ k\\
&\texttt{tail}\ k\ \ xs \qquad\qquad\qquad = \boxed{\{\ \}}
\end{aligned}
$$

Dependent pattern matching

$$
\begin{array}{l}
\textbf{data}\ \mathtt{Vec}\ (A : \mathtt{Type}) : \mathbb{N} \to \mathtt{Type}\ \textbf{where} \\
\quad \mathtt{nil}\ \ : \mathtt{Vec}\ A\ \mathtt{zero} \\
\quad \mathtt{cons} : (n : \mathbb{N}) \to A \to \mathtt{Vec}\ A\ n \to \mathtt{Vec}\ A\ (\mathtt{suc}\ n)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathtt{tail} : (k : \mathbb{N}) \to \mathtt{Vec}\ A\ (\mathtt{suc}\ k) \to \mathtt{Vec}\ A\ k} \\
\mathtt{tail}\ k\ \ \mathtt{nil} & = \boxed{\{\ \}} & \texttt{-- suc k = zero} \\
\mathtt{tail}\ k\ \ (\mathtt{cons}\ n\ x\ xs) & = \boxed{\{\ \}} & \texttt{-- suc k = suc n}
\end{array}
$$

Dependent pattern matching

$$\begin{array}{l}
\textbf{data}\ \texttt{Vec}\ (A : \texttt{Type}) : \mathbb{N} \to \texttt{Type}\ \textbf{where} \\
\quad \texttt{nil}\ \ \ : \texttt{Vec}\ A\ \texttt{zero} \\
\quad \texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)
\end{array}$$

$$\begin{array}{l}
\texttt{tail} : (k : \mathbb{N}) \to \texttt{Vec}\ A\ (\texttt{suc}\ k) \to \texttt{Vec}\ A\ k \\
\texttt{tail}\ k\ \ \texttt{nil} \qquad\qquad\quad = \boxed{\{\ \}}\ \ \texttt{-- impossible} \\
\texttt{tail}\ k\ \ (\texttt{cons}\ n\ x\ xs) = \boxed{\{\ \}}\ \ \texttt{-- suc k = suc n}
\end{array}$$

Dependent pattern matching

$$\textbf{data}\ \texttt{Vec}\ (A : \texttt{Type}) : \mathbb{N} \to \texttt{Type}\ \textbf{where}$$
$$\quad \texttt{nil}\ \ : \texttt{Vec}\ A\ \texttt{zero}$$
$$\quad \texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)$$

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

$$\texttt{tail}\ k\ \ (\texttt{cons}\ n\ x\ xs) = \boxed{\{\ \}}\ \ \texttt{-- suc k = suc n}$$

Dependent pattern matching

$$
\begin{array}{l}
\textbf{data}\ \texttt{Vec}\ (A : \texttt{Type}) : \mathbb{N} \to \texttt{Type}\ \textbf{where} \\
\quad \texttt{nil}\quad : \texttt{Vec}\ A\ \texttt{zero} \\
\quad \texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)
\end{array}
$$

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

$$\texttt{tail}\ k\ \ (\texttt{cons}\ n\ x\ xs) = \boxed{\{\ \}}\ \ \texttt{--}\qquad \texttt{k =}\qquad \texttt{n}$$

Dependent pattern matching

$$
\begin{array}{l}
\textbf{data}\ \mathtt{Vec}\ (A : \mathtt{Type}) : \mathbb{N} \to \mathtt{Type}\ \textbf{where} \\
\quad \mathtt{nil}\ \ \ : \mathtt{Vec}\ A\ \mathtt{zero} \\
\quad \mathtt{cons} : (n : \mathbb{N}) \to A \to \mathtt{Vec}\ A\ n \to \mathtt{Vec}\ A\ (\mathtt{suc}\ n)
\end{array}
$$

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

$$\mathtt{tail}\ .n\ (\mathtt{cons}\ n\ x\ xs) = \boxed{\{\ \}}$$

Dependent pattern matching

$$
\begin{array}{l}
\textbf{data}\ \texttt{Vec}\ (A : \texttt{Type}) : \mathbb{N} \to \texttt{Type}\ \textbf{where} \\
\quad \texttt{nil}\ \ : \texttt{Vec}\ A\ \texttt{zero} \\
\quad \texttt{cons} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (\texttt{suc}\ n)
\end{array}
$$

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

$$\texttt{tail}\ .n\ (\texttt{cons}\ n\ x\ xs) = xs$$

Matching on the identity type

$$\mathbf{data}\ \_\equiv\_\ (x : A) : A \to \mathtt{Type}\ \mathbf{where}$$
$$\quad \mathtt{refl} : x \equiv x$$

$$\mathtt{trans} : (x\ y\ z : A) \to (x \equiv y) \to (y \equiv z) \to (x \equiv z)$$
$$\mathtt{trans}\ x\ y\ z\ p\ \ q\ \ = \boxed{\{\ \}}$$

Matching on the identity type

$$\textbf{data}\ \_\equiv\_\ (x : A) : A \to \mathtt{Type}\ \textbf{where}$$
$$\quad \mathtt{refl} : x \equiv x$$

$$\mathtt{trans} : (x\ y\ z : A) \to (x \equiv y) \to (y \equiv z) \to (x \equiv z)$$
$$\mathtt{trans}\ x\ .x\ z\ \ \mathtt{refl}\ q \qquad = \boxed{\{\ \}}$$

Matching on the identity type

$\textbf{data}\ \_\equiv\_\ (x : A) : A \to \texttt{Type}\ \textbf{where}$

$\quad \texttt{refl} : x \equiv x$

$\texttt{trans} : (x\ y\ z : A) \to (x \equiv y) \to (y \equiv z) \to (x \equiv z)$

$\texttt{trans}\ x\ .x\ .x\ \texttt{refl}\ \texttt{refl} = \{\ \}$

Matching on the identity type

$\textbf{data}\ \_\equiv\_\ (x : A) : A \to \texttt{Type}\ \textbf{where}$
$\quad \texttt{refl} : x \equiv x$

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

Pattern matching implies uniqueness of identity proofs

$$\mathtt{UIP} : (x : A)(e : x \equiv_A x) \to e \equiv_{x \equiv_A x} \mathtt{refl}$$
$$\mathtt{UIP}\ x\ e \qquad = \boxed{\{\ \}}$$

Pattern matching implies uniqueness of identity proofs

$$\mathtt{UIP} : (x : A)(e : x \equiv_A x) \to e \equiv_{x \equiv_A x} \mathtt{refl}$$
$$\mathtt{UIP}\ x\ \mathtt{refl} = \boxed{\{\ \}}$$

Pattern matching implies uniqueness of identity proofs

$$\texttt{UIP} : (x : A)(e : x \equiv_A x) \to e \equiv_{x \equiv_A x} \texttt{refl}$$
$$\texttt{UIP}\ x\ \texttt{refl} = \texttt{refl}$$

We don't always want to assume UIP

UIP is incompatible with univalence

We don't always want to assume UIP

UIP is incompatible with univalence:

- UIP implies $\texttt{coerce}\ e\ \texttt{true} = \texttt{true}$ for all $e : \texttt{Bool} \equiv \texttt{Bool}$

We don't always want to assume UIP

UIP is incompatible with univalence:

- UIP implies $\texttt{coerce}\ e\ \texttt{true} = \texttt{true}$ for all $e : \texttt{Bool} \equiv \texttt{Bool}$
- Univalence gives $\texttt{swap} : \texttt{Bool} \equiv \texttt{Bool}$ such that $\texttt{coerce}\ \texttt{swap}\ \texttt{true} = \texttt{false}$

We don't always want to assume UIP

UIP is incompatible with univalence:

- UIP implies $\texttt{coerce}\ e\ \texttt{true} = \texttt{true}$ for all $e : \texttt{Bool} \equiv \texttt{Bool}$
- Univalence gives $\texttt{swap} : \texttt{Bool} \equiv \texttt{Bool}$ such that $\texttt{coerce}\ \texttt{swap}\ \texttt{true} = \texttt{false}$

hence $\texttt{true} = \texttt{false}$!

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!

Dependent pattern matching

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.

Unification problems as telescopes

A **unification problem** consists of

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

Unification problems as telescopes

A **unification problem** consists of

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

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

$$(x_1 : A_1)(x_2 : A_2) \ldots$$
$$(e_1 : u_1 \equiv_{B_1} v_1)(e_2 : u_2 \equiv_{B_2} v_2) \ldots$$

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

Unification problems as telescopes

A **unification problem** consists of

1. Flexible variables Γ
2. Equations $u_1 = v_1 : B_1, \ldots$

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

$$\begin{array}{l} \Gamma \\ (e_1 : u_1 \equiv_{B_1} v_1)(e_2 : u_2 \equiv_{B_2} v_2) \ldots \end{array}$$

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

Unification problems as telescopes

A **unification problem** consists of

1. 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. $(k : \mathbb{N})(n : \mathbb{N})(e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ n)$

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. $f : () \to (n : \mathbb{N})(e : n \equiv_{\mathbb{N}} \texttt{zero})$

Evidence of unification

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

Evidence of unification

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

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

Evidence of unification

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

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

Evidence of unification

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

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

$\Longrightarrow$ Unification is guaranteed to respect $\equiv$!

Three valid unifiers

$$f_1 : (k : \mathbb{N}) \to (k\ n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)$$
$$f_1\ k = k; k; \mathtt{refl}$$

$$f_2 : () \to (k\ n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)$$
$$f_2\ () = \mathtt{zero}; \mathtt{zero}; \mathtt{refl}$$

$$f_3 : (k\ n : \mathbb{N}) \to (k\ n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)$$
$$f_3\ k\ n = k; k; \mathtt{refl}$$

What is a most general unifier?

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 $\sigma' = \sigma \circ \nu$.

What is a most general unifier?

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.

What is a most general unifier?

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

What is a most general unifier?

$$
\begin{tikzcd}
\Gamma'' \arrow[drr, bend left, "f'"] & & \\
\Gamma' \arrow[rr, "f"] & & \Gamma(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v})
\end{tikzcd}
$$

What is a most general unifier?

$$\begin{tikzcd}
\Gamma'' \arrow[d, dashed, "h"'] \arrow[dr, bend left, "f'"] & \\
\Gamma' \arrow[r, "f"] & \Gamma(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v})
\end{tikzcd}$$

What is a most general unifier?

$$\begin{tikzcd}
\Gamma'' \arrow[d, dashed, "h"'] \arrow[drr, bend left, "f'"] & & \\
\Gamma' \arrow[rr, "f"] & & \Gamma(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v}) \arrow[ll, bend right, "g_1"']
\end{tikzcd}$$

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

What is a most general unifier?

$$\begin{tikzcd}
\Gamma'' \arrow[d, dashed, "h"'] \arrow[dr, bend left, "f'"] & \\
\Gamma' \arrow[r, "f"] & \Gamma(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v}) \arrow[l, bend right, "g_1"'] \arrow[l, bend left, "g_2"]
\end{tikzcd}$$

$$\begin{array}{lcl}
(\forall f' : h \text{ exists}) & \Leftrightarrow & f \text{ has a } \textit{right inverse } g_1 \\
(\forall f' : h \text{ is unique}) & \Leftrightarrow & f \text{ has a } \textit{left inverse } g_2
\end{array}$$

What is a most general unifier?

$$\begin{tikzcd}
\Gamma' \arrow[r, "f"] & \Gamma(\bar{e} : \bar{u} \equiv_{\Delta} \bar{v}) \arrow[l, bend right, "g_1"'] \arrow[l, bend left, "g_2"]
\end{tikzcd}$$

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

Most general unifiers are equivalences!

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

Example of unification

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

Example of unification

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

$$\wr\!\!\mid$$

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

Example of unification

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

$$\wr|$$

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

$$\wr|$$

$$(k : \mathbb{N})$$

Example of unification

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

$$\wr|$$

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

$$\wr|$$

$$(k : \mathbb{N})$$

$$f : (k : \mathbb{N}) \to (k\ n : \mathbb{N})(e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ n)$$
$$f\ k = k; k; \mathtt{refl}$$

The solution rule

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

The deletion rule

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

The injectivity rule

$$\texttt{injectivity}_{\texttt{suc}} :$$
$$(e : \texttt{suc}\ x \equiv_{\mathbb{N}} \texttt{suc}\ y) \simeq (e' : x \equiv_{\mathbb{N}} y)$$

Negative unification rules

A *negative unification rule* applies to impossible equations, e.g. $\texttt{suc}\ x = \texttt{zero}$.

Negative unification rules

A *negative unification rule* applies to impossible equations, e.g. $\texttt{suc}\ x = \texttt{zero}$.

This can be represented by an equivalence:

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

where $\bot$ is the **empty type**.

The conflict rule

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

The cycle rule

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

Heterogeneous equations

Problem: What is the type of e_2?

$$(e : (0, \texttt{nil}) \equiv_{\Sigma_{n:\mathbb{N}} \texttt{Vec}\ A\ n} (1, \texttt{cons}\ 0\ x\ xs))$$

$$\wr\!|$$

$$(e_1 : 0 \equiv_{\mathbb{N}} 1)(e_2 : \texttt{nil} \equiv_{\texttt{Vec}\ A\ ???} \texttt{cons}\ 0\ x\ xs)$$

Heterogeneous equations

Problem: What is the type of e_2?

Solution: keep track of dependencies by introducing a new variable for each equation

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

$$\wr|$$

$$(e_1 : 0 \equiv_{\mathbb{N}} 1)(e_2 : \mathtt{nil} \equiv_{\mathtt{Vec}\ A\ e_1} \ \mathtt{cons}\ 0\ x\ xs)$$

Heterogeneous equations

Problem: What is the type of e_2?
Solution: keep track of dependencies by introducing a new variable for each equation

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

This is called a *telescopic equality*.

Be careful with heterogeneous equations!

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

Be careful with heterogeneous equations!

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

$$\wr|$$

$$(e_1 : \texttt{Bool} \equiv_{\texttt{Type}} \texttt{Bool})(e_2 : \texttt{true} \equiv_{e_1} \texttt{false})$$

Be careful with heterogeneous equations!

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

$$\wr|$$

$$(e_1 : \mathtt{Bool} \equiv_{\mathtt{Type}} \mathtt{Bool})(e_2 : \mathtt{true} \equiv_{e_1} \mathtt{false})$$

$$\wr|$$

$$\perp$$

Be careful with heterogeneous equations!

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

$$\wr\|$$

$$(e_1 : \texttt{Bool} \equiv_{\texttt{Type}} \texttt{Bool})(e_2 : \texttt{true} \equiv_{e_1} \texttt{false})$$

$$\not\cong$$

$$\perp$$

The `conflict` rule does not apply!

Be careful with heterogeneous equations!

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

Be careful with heterogeneous equations!

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

$$\wr|$$

$$(e_1 : \texttt{Bool} \equiv_{\texttt{Type}} \texttt{Bool})(e_2 : \texttt{true} \equiv_{\texttt{Bool}} \texttt{false})$$

Be careful with heterogeneous equations!

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

$$\wr|$$

$$(e_1 : \mathtt{Bool} \equiv_{\mathtt{Type}} \mathtt{Bool})(e_2 : \mathtt{true} \equiv_{\mathtt{Bool}} \mathtt{false})$$

$$\wr|$$

$$\bot$$

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:

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

Injectivity for indexed data

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

$$
\begin{gathered}
(e_1 : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ n) \\
(e_2 : \mathtt{cons}\ k\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys) \\
\wr| \\
(e_1' : k \equiv_{\mathbb{N}} n)(e_2' : x \equiv_A y) \\
(e_3' : xs \equiv_{\mathtt{Vec}\ A\ e_1} ys)
\end{gathered}
$$

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

Solving unsolvable equations

data $\texttt{Im}\ (f : A \to B) : B \to \texttt{Type}$ **where**

$\quad \texttt{image} : (x : A) \to \texttt{Im}\ f\ (f\ x)$

Solving unsolvable equations

$$\textbf{data}\ \texttt{Im}\ (f : A \to B) : B \to \texttt{Type}\ \textbf{where}$$
$$\quad \texttt{image} : (x : A) \to \texttt{Im}\ f\ (f\ x)$$

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

Solving unsolvable equations

data $\texttt{Im}\ (f : A \to B) : B \to \texttt{Type}$ **where**

$\quad \texttt{image} : (x : A) \to \texttt{Im}\ f\ (f\ x)$

$$(x_1\ x_2 : A)(e_1 : f\ x_1 \equiv_B f\ x_2)$$
$$(e_2 : \texttt{image}\ x_1 \equiv_{\texttt{Im}\ f\ e_1} \texttt{image}\ x_2)$$
$$\wr|$$
$$(x_1\ x_2 : A)(e : x_1 \equiv_A x_2)$$

Solving unsolvable equations

data $\texttt{Im}\ (f : A \to B) : B \to \texttt{Type}$ **where**

$\quad \texttt{image} : (x : A) \to \texttt{Im}\ f\ (f\ x)$

$$(x_1\ x_2 : A)(e_1 : f\ x_1 \equiv_B f\ x_2)$$
$$(e_2 : \texttt{image}\ x_1 \equiv_{\texttt{Im}\ f\ e_1} \texttt{image}\ x_2)$$
$$\wr|$$
$$(x_1\ x_2 : A)(e : x_1 \equiv_A x_2)$$
$$\wr|$$
$$(x_1 : A)$$

Dependent pattern matching

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

What if the indices are not fully general?

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

What if the indices are not fully general?

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

$$\wr|$$

$$(e_1 : \texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n)$$

$$(e_2 : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ e_1} \texttt{cons}\ n\ y\ ys)$$

$$(p : e_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl})$$

What if the indices are not fully general?

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

$$\wr|$$

$$(e_1 : \mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n)$$

$$(e_2 : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ e_1} \mathtt{cons}\ n\ y\ ys)$$

$$(p : e_1 \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})$$

$$\wr|$$

$$(e'_1 : n \equiv_{\mathbb{N}} n)(e'_2 : x \equiv_A y)(e'_3 : xs \equiv_{\mathtt{Vec}\ A\ e'_1} ys)$$

$$(p : \mathtt{cong}\ \mathtt{suc}\ e'_1 \equiv_{\mathtt{suc}\ n \equiv_{\mathbb{N}} \mathtt{suc}\ n} \mathtt{refl})$$

What if the indices are not fully general?

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

$$\wr|$$

$$(e_1 : \texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n)$$

$$(e_2 : \texttt{cons}\ n\ x\ xs \equiv_{\texttt{Vec}\ A\ e_1} \texttt{cons}\ n\ y\ ys)$$

$$(p : e_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl})$$

$$\wr|$$

$$(e'_1 : n \equiv_{\mathbb{N}} n)(e'_2 : x \equiv_A y)(e'_3 : xs \equiv_{\texttt{Vec}\ A\ e'_1} ys)$$

$$(p : \texttt{cong}\ \texttt{suc}\ e'_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl})$$

Higher-dimensional equations

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

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

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

let's first consider

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

Step 2: solve lower-dimensional equations

This gives us an equivalence f of type

$$\begin{gathered}(k : \mathbb{N})(u : A)(us : \mathtt{Vec}\ A\ k) \\ (e : \mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ n) \\ \wr\!\mid \\ (u : A)(us : \mathtt{Vec}\ A\ n)\end{gathered}$$

Step 3: lift unifier to higher dimension

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

$$
\begin{gathered}
(e_1 : n \equiv_{\mathbb{N}} n)(e_2 : x \equiv_A y) \\
(e_3 : xs \equiv_{\texttt{Vec}\ A\ e_1} ys) \\
(p : \texttt{cong}\ \texttt{suc}\ e_1 \equiv_{\texttt{suc}\ n \equiv_{\mathbb{N}} \texttt{suc}\ n} \texttt{refl}) \\
\wr| \\
(e_2 : x \equiv_A y)(e_3 : xs \equiv_{\texttt{Vec}\ A\ n} ys)
\end{gathered}
$$

Final result of steps 1-3

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

$$\wr|$$

$$\left(e_2 : x \equiv_A y\right)\left(e_3 : xs \equiv_{\texttt{Vec}\ A\ n} ys\right)$$

Final result of steps 1-3

$$\left(e : \mathtt{cons}\ n\ x\ xs \equiv_{\mathtt{Vec}\ A\ (\mathtt{suc}\ n)} \mathtt{cons}\ n\ y\ ys\right)$$

$$\wr|$$

$$\left(e_2 : x \equiv_A y\right)\left(e_3 : xs \equiv_{\mathtt{Vec}\ A\ n} ys\right)$$

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

Lifting equivalences: (mostly) general case

Theorem. If we have an equivalence f of type

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

we can construct $f^{\uparrow}$ of type

$$(e : u \equiv_A v)(p : \texttt{cong}\ b_1\ e \equiv_{r \equiv_{B\ e} s} \texttt{cong}\ b_2\ e)$$
$$\wr|$$
$$(e' : f\ u\ r \equiv_C f\ v\ s)$$

Dependent pattern matching

Proof-relevant unification

Higher-dimensional unification

Back to eliminators

Desugaring pattern matching

1. Translate the definition to a case tree.

Desugaring pattern matching

1. Translate the definition to a case tree.
2. Translate each case split,
   using basic `case`$_{\mathtt{D}}$-analysis.

Desugaring pattern matching

1. Translate the definition to a case tree.
2. Translate each case split, using basic $\texttt{case}_{\texttt{D}}$-analysis.
3. Solve equations in each subtree, using specialization by unification.

Desugaring pattern matching

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

From pattern matching . . .

$$
\begin{array}{l}
\textbf{data}\ \_ \leq \_ : \mathbb{N} \to \mathbb{N} \to \mathtt{Type}\ \textbf{where} \\
\quad \mathtt{lz} : (l : \mathbb{N}) \to \mathtt{zero} \leq l \\
\quad \mathtt{ls} : (k\ l : \mathbb{N}) \to k \leq l \to \mathtt{suc}\ k \leq \mathtt{suc}\ l
\end{array}
$$

$$
\begin{array}{l}
\mathtt{antisym} : (x\ y : \mathbb{N}) \to x \leq y \to y \leq x \to x \equiv y \\
\mathtt{antisym}\ .\mathtt{zero}\quad .\mathtt{zero}\quad (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor)\ (\mathtt{lz}\ \lfloor \mathtt{zero} \rfloor) = \mathtt{refl} \\
\mathtt{antisym}\ .(\mathtt{suc}\ k)\ .(\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ u)\quad (\mathtt{ls}\ .l\ .k\ v) \\
\qquad = \mathtt{cong}\ \mathtt{suc}\ (\mathtt{antisym}\ k\ l\ u\ v)
\end{array}
$$

. . . to a case tree . . .

$$
x\ y\ \underline{u}\ v \begin{cases}
.\mathtt{zero}\ y\ (\mathtt{lz}\ .y)\ \underline{u} \\
\quad \left\{ .\mathtt{zero}\ .\mathtt{zero}\ (\mathtt{lz}\ .\mathtt{zero})\ (\mathtt{lz}\ .\mathtt{zero}) \mapsto \mathtt{refl} \right. \\
.(\mathtt{suc}\ k)\ .(\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ u)\ \underline{v} \\
\quad \begin{cases} .(\mathtt{suc}\ k)\ .(\mathtt{suc}\ l)\ (\mathtt{ls}\ k\ l\ u)\ (\mathtt{ls}\ .l\ .k\ v) \\ \qquad \mapsto \mathtt{cong}\ \mathtt{suc}\ (\mathtt{antisym}\ k\ l\ u\ v) \end{cases}
\end{cases}
$$

. . . to eliminators.

$$
\begin{array}{l}
\mathtt{antisym} : (x\ y : \mathbb{N}) \to x \leq y \to y \leq x \to x \equiv_{\mathbb{N}} y \\
\mathtt{antisym} = \mathtt{elim}_{\leq}\ (\lambda x\, y\, u.\, y \leq x \to x \equiv_{\mathbb{N}} y) \\
\quad (\lambda l\, v.\, \mathtt{elim}_{\leq}\ (\lambda y\, x\, v.\, x \equiv_{\mathbb{N}} \mathtt{zero} \to x \equiv_{\mathbb{N}} y) \\
\qquad (\lambda x\, e.\, e) \\
\qquad (\lambda l\, k\, \_\, \_\, e.\, \mathtt{elim}_{\bot}\ (\mathtt{suc}\ k \equiv_{\mathbb{N}} \mathtt{suc}\ l)\ (\mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ k)\ \mathtt{zero}\ e)) \\
\qquad l\ \mathtt{zero}\ v\ \mathtt{refl}) \\
\quad (\lambda k\, l\, \_\, H\, v.\, \mathtt{cong}\ \mathtt{suc}\ (H \\
\qquad (\mathtt{elim}_{\leq}\ (\lambda x\, y\, \_.\, x \equiv_{\mathbb{N}} \mathtt{suc}\ l \to u \equiv_{\mathbb{N}} \mathtt{suc}\ k \to l \leq k) \\
\qquad\quad (\lambda l'\, e\, \_.\, \mathtt{elim}_{\bot}\ (l \leq k)\ (\mathtt{noConf}_{\mathbb{N}}\ \mathtt{zero}\ (\mathtt{suc}\ l)\ e)) \\
\qquad\quad (\lambda k'\, l'\, v'\, \_\, e_1\, e_2.\, \mathtt{subst}\ (\lambda n.\ n \leq k) \\
\qquad\qquad (\mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ k')\ (\mathtt{suc}\ l)\ e_1) \\
\qquad\qquad (\mathtt{subst}\ (\lambda m.\ k' \leq m)\ (\mathtt{noConf}_{\mathbb{N}}\ (\mathtt{suc}\ l')\ (\mathtt{suc}\ k)\ e_2)\ v')) \\
\qquad\quad (\mathtt{suc}\ l)\ (\mathtt{suc}\ k)\ v\ \mathtt{refl}\ \mathtt{refl})))
\end{array}
$$