Depending on equations

A proof-relevant framework for unification in dependent type theory

Jesper Cockx

DistriNet – KU Leuven

3 September 2017

Unification for dependent types

Unification is used for many purposes:

logic programming, type inference, term rewriting, automated theorem proving, natural language processing, . . .

This talk:

checking definitions by dependent pattern matching

Disclaimer

My work is on dependently typed languages,
I know little about unification.

Disclaimer

My work is on dependently typed languages,
I know little about unification.

This talk is about **first-order unification**:

$$(\texttt{suc}\ x = \texttt{suc}\ y) \Rightarrow (x = y) \xrightarrow{x:=y} \text{OK}$$

Disclaimer

My work is on dependently typed languages,
I know little about unification.

This talk is about **first-order unification**:

$$(\texttt{suc}\ x = \texttt{suc}\ y) \Rightarrow (x = y) \xRightarrow{x := y} \text{OK}$$

$$(\texttt{suc}\ x = \texttt{zero}) \Rightarrow \bot$$

Disclaimer

My work is on dependently typed languages,
I know little about unification.

This talk is about **first-order unification**:

$$(\texttt{suc}\ x = \texttt{suc}\ y) \Rightarrow (x = y) \xRightarrow{x:=y} \mathsf{OK}$$

$$(\texttt{suc}\ x = \texttt{zero}) \Rightarrow \bot$$

...but there will be types everywhere!

Dependent types: the 'big five'

During this presentation, we'll spot:

- Dependent functions: $(x : A) \to B\ x$

Dependent types: the ‘big five’

During this presentation, we’ll spot:

- Dependent functions: $(x : A) \to B\ x$
- Indexed datatypes: `Vec` $A\ n$, ...

Dependent types: the 'big five'

During this presentation, we'll spot:

- Dependent functions: $(x : A) \to B\ x$
- Indexed datatypes: $\texttt{Vec}\ A\ n, \ldots$
- Identity types: $x \equiv_A y$

Dependent types: the 'big five'

During this presentation, we'll spot:

- Dependent functions: $(x : A) \to B\ x$
- Indexed datatypes: $\texttt{Vec}\ A\ n$, ...
- Identity types: $x \equiv_A y$
- Universes: $\texttt{Type}_i$

Dependent types: the 'big five'

During this presentation, we'll spot:

- Dependent functions: $(x : A) \to B\ x$
- Indexed datatypes: $\texttt{Vec}\ A\ n, \ldots$
- Identity types: $x \equiv_A y$
- Universes: $\texttt{Type}_i$
- Univalence: $(A \equiv B) \simeq (A \simeq B)$

Dependent types: the ‘big five’

During this presentation, we’ll spot:

- Dependent functions: $(x : A) \to B\ x$
- Indexed datatypes: $\texttt{Vec}\ A\ n$, ...
- Identity types: $x \equiv_A y$
- Universes: $\texttt{Type}_i$
- Univalence: $(A \equiv B) \simeq (A \simeq B)$

and see how they interact with unification!

Depending on equations

Checking dependently typed programs

Unification in dependent type theory

Unification of dependently typed terms

Depending on equations

Checking dependently typed programs

Unification in dependent type theory

Unification of dependently typed terms

Why use dependent types?

With dependent types, you can . . .

Why use dependent types?

With dependent types, you can …

… guarantee that a program matches its specification

Why use dependent types?

With dependent types, you can …

- … guarantee that a program matches its specification
- … use the same language for writing programs and proofs

Why use dependent types?

With dependent types, you can ...

- ... guarantee that a program matches its specification
- ... use the same language for writing programs and proofs
- ... develop programs and proofs interactively

Dependent types

Per
Martin-Löf

A **dependent type** is a family of types, depending on a term of a **base type**.

Dependent types

Per Martin-Löf

A **dependent type** is a family of types, depending on a term of a **base type**.

e.g. `Vec` A n is the type of vectors of length n.

The Agda language

Agda is a purely functional language

The Agda language

Agda is a purely functional language

. . . with a strong, static type system

The Agda language

Agda is a purely functional language

... with a strong, static type system

... for writing programs and proofs

The Agda language

Agda is a purely functional language

. . . with a strong, static type system

. . . for writing programs and proofs

. . . with datatypes and pattern matching

The Agda language

Agda is a purely functional language

... with a strong, static type system

... for writing programs and proofs

... with datatypes and pattern matching

... with first-class dependent types

The Agda language

Agda is a purely functional language

... with a strong, static type system

... for writing programs and proofs

... with datatypes and pattern matching

... with first-class dependent types

... with support for interactive development

The Agda language

Agda is a purely functional language

... with a strong, static type system

... for writing programs and proofs

... with datatypes and pattern matching

... with first-class dependent types

... with support for interactive development

All examples are (mostly) valid Agda code!

Using dependent types

With dependent types, we can give more precise types to our programs:

$$\texttt{replicate} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n$$

Using dependent types

With dependent types, we can give more precise types to our programs:

$$\texttt{replicate} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n$$

$$\Rightarrow \texttt{replicate}\ 10\ \text{'a'} : \texttt{Vec}\ \texttt{Char}\ 10$$

Using dependent types

With dependent types, we can give more precise types to our programs:

$$\texttt{replicate} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n$$

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

Using dependent types

With dependent types, we can give more precise types to our programs:

$$\texttt{replicate} : (n : \mathbb{N}) \to A \to \texttt{Vec}\ A\ n$$

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

$$\texttt{append} : (m\ n : \mathbb{N}) \to \texttt{Vec}\ A\ m \to \texttt{Vec}\ A\ n \to \texttt{Vec}\ A\ (m + n)$$

Simple pattern matching

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

Simple pattern matching

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

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

Simple pattern matching

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

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

Simple pattern matching

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

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

Simple pattern matching

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

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

Simple pattern matching

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

$$\mathtt{minimum} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathtt{minimum}\ \mathtt{zero}\quad y \qquad = \mathtt{zero}$$
$$\mathtt{minimum}\ (\mathtt{suc}\ x)\ \mathtt{zero} = \mathtt{zero}$$
$$\mathtt{minimum}\ (\mathtt{suc}\ x)\ (\mathtt{suc}\ y) = \mathtt{suc}\ (\mathtt{minimum}\ x\ y)$$

Dependent pattern matching

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

Dependent pattern matching

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

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

Dependent pattern matching

```
data Vec (A : Type) : ℕ → Type where
  nil  : Vec A zero
  cons : (n : ℕ) → A → Vec A n → Vec A (suc n)

tail : (k : ℕ) → Vec A (suc k) → Vec A k
tail k  nil             = { } -- suc k = zero
tail k  (cons n x xs)   = { } -- suc k = suc n
```

Dependent pattern matching

```
data Vec (A : Type) : ℕ → Type where
  nil  : Vec A zero
  cons : (n : ℕ) → A → Vec A n → Vec A (suc n)

tail : (k : ℕ) → Vec A (suc k) → Vec A k
tail k  nil            = { }  -- impossible
tail k  (cons n x xs)  = { }  -- suc k = suc n
```

Dependent pattern matching

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

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

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

Dependent pattern matching

data Vec $(A : \texttt{Type}) : \mathbb{N} \to$ 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 k (cons n x xs) = { } -- k = n

Dependent pattern matching

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

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

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

Dependent pattern matching

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

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

tail $.n$ (cons $n\ x\ xs$) $=\ xs$

Specialization by unification

Agda uses unification to:

- eliminate impossible cases
- specialize the result type

Specialization by unification

Agda uses unification to:

- eliminate impossible cases
- specialize the result type

The output of unification can change Agda's notion of equality!

Specialization by unification

Agda uses unification to:

- eliminate impossible cases
- specialize the result type

The output of unification can change Agda's notion of equality!

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

Depending on equations

Checking dependently typed programs

Unification in dependent type theory

Unification of dependently typed terms

Q: What is the fastest way to start a fight between type theorists?

Q: What is the fastest way to start a fight between type theorists?

A: Mention the topic of equality.

The identity type

$$x \equiv_A y$$

. . . a dependent type depending on $x, y : A$.

The identity type

$$x \equiv_A y$$

…a dependent type depending on $x, y : A$.

…type theory's built-in notion of equality.

The identity type

$$x \equiv_A y$$

...a dependent type depending on $x, y : A$.

...type theory's built-in notion of equality.

...the type of **proofs** that $x = y$.

Operations on the identity type

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

Operations on the identity type

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

$\texttt{sym} \quad : x \equiv_A y \to y \equiv_A x$

Operations on the identity type

$$\begin{aligned}
\texttt{refl} &: x \equiv_A x \\
\texttt{sym} &: x \equiv_A y \to y \equiv_A x \\
\texttt{trans} &: x \equiv_A y \to y \equiv_A z \to x \equiv_A z
\end{aligned}$$

Operations on the identity type

$$\begin{aligned}
\texttt{refl} &: x \equiv_A x \\
\texttt{sym} &: x \equiv_A y \to y \equiv_A x \\
\texttt{trans} &: x \equiv_A y \to y \equiv_A z \to x \equiv_A z \\
\texttt{cong}\ f &: x \equiv_A y \to f\ x \equiv_B f\ y
\end{aligned}$$

Operations on the identity type

$$\begin{aligned}
\texttt{refl} &: x \equiv_A x \\
\texttt{sym} &: x \equiv_A y \to y \equiv_A x \\
\texttt{trans} &: x \equiv_A y \to y \equiv_A z \to x \equiv_A z \\
\texttt{cong}\ f &: x \equiv_A y \to f\ x \equiv_B f\ y \\
\texttt{subst}\ P &: x \equiv_A y \to P\ x \to P\ y
\end{aligned}$$

Unification problems as telescopes

A **unification problem** consists of

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

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**:

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

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 $\bar{u} = \bar{v} : \Delta$

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

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

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

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

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

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

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

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

Most general unifiers

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

Most general unifiers

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

This is quite difficult to translate to type theory directly…

Most general unifiers

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

This is quite difficult to translate to type theory directly...

Intuition: if $f : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ is MGU, we can go back from $\Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ to Γ' without losing any information.

Equivalences

A function $f : A \to B$ is an **equivalence** if it has both a left and a right inverse:

$$\texttt{isLinv} : (x : A) \to g_1\ (f\ x) \equiv_A x$$
$$\texttt{isRinv} : (y : B) \to f\ (g_2\ y) \equiv_B y$$

In this case, we write $f : A \simeq B$.

Most general unifiers are equivalences!

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

Example of unification

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

Example of unification

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

$$\wr|$$

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

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

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

$$\wr|$$

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

$$\wr|$$

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

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

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. `suc` $x =$ `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

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

The cycle rule

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

Unifiers as equivalences

By requiring **unifiers** to be **equivalences**:

- we exclude bad unification rules
- we can safely introduce new rules

Unifiers as equivalences

By requiring **unifiers** to be **equivalences**:

- we exclude bad unification rules
- we can safely introduce new rules

Next, we'll explore how this idea can help us.

Any questions so far?

Depending on equations

Checking dependently typed programs

Unification in dependent type theory

Unification of dependently typed terms

Time for the interesting bits!

- Equations between types
- Heterogeneous equations
- Equations on indexed datatypes
- Equations between equations

Equations between types

Types are first-class terms of type $\texttt{Type}$:
$\texttt{Bool} : \texttt{Type}$, $\mathbb{N} : \texttt{Type}$, $\mathbb{N} \to \mathbb{N} : \texttt{Type}$, $\ldots$

Equations between types

Types are first-class terms of type $\texttt{Type}$:
$\texttt{Bool} : \texttt{Type}$, $\mathbb{N} : \texttt{Type}$, $\mathbb{N} \to \mathbb{N} : \texttt{Type}$, ...

We can form equations between types,
e.g. $\texttt{Bool} \equiv_{\texttt{Type}} \texttt{Bool}$.

Equations between types

Types are first-class terms of type $\texttt{Type}$:
$\texttt{Bool} : \texttt{Type}$, $\mathbb{N} : \texttt{Type}$, $\mathbb{N} \to \mathbb{N} : \texttt{Type}$, ...

We can form equations between types,
e.g. $\texttt{Bool} \equiv_{\texttt{Type}} \texttt{Bool}$.

Q: Can we apply the `deletion` rule?

Equations between types

Types are first-class terms of type $\texttt{Type}$:
$\texttt{Bool} : \texttt{Type}$, $\mathbb{N} : \texttt{Type}$, $\mathbb{N} \to \mathbb{N} : \texttt{Type}$, ...

We can form equations between types,
e.g. $\texttt{Bool} \equiv_{\texttt{Type}} \texttt{Bool}$.

Q: Can we apply the `deletion` rule?

A: Depends on which type theory we use!

The univalence axiom (2009)

Vladimir
Voevodsky

The univalence axiom (2009)

Vladimir Voevodsky

“Isomorphic types can be identified.”

The univalence axiom (2009)

Vladimir Voevodsky

"Isomorphic types can be identified."

$$(A \equiv B) \simeq (A \simeq B)$$

The univalence axiom (2009)

`Bool` is equal to `Bool` in two ways:

`Bool`

`true`	`false`

The univalence axiom (2009)

`Bool` is equal to `Bool` in two ways:

`Bool`

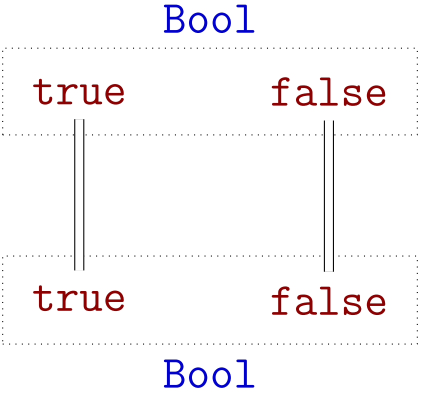
`true` `false`

`true` `false`

`Bool`

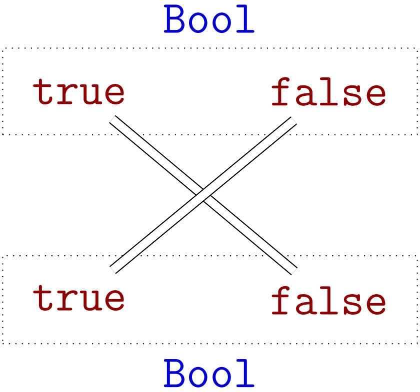
The univalence axiom (2009)

Bool is equal to Bool in two ways:



The univalence axiom (2009)

Bool is equal to Bool in two ways:



Limiting the deletion rule

The `deletion` rule does not always hold: there might be multiple proofs of $x \equiv_A x$.

E.g. `Bool` $\equiv_{\texttt{Type}}$ `Bool` has two elements.

Limiting the deletion rule

The `deletion` rule does not always hold: there might be multiple proofs of $x \equiv_A x$.

E.g. `Bool` $\equiv_{\texttt{Type}}$ `Bool` has two elements.

We cannot use `deletion` in this case!

Heterogeneous equations

$\Sigma_{n:\mathbb{N}}\texttt{Vec}\ A\ n$ is the type of pairs (n, xs) where $n : \mathbb{N}$ and $xs : \texttt{Vec}\ A\ n$.

Heterogeneous equations

$\Sigma_{n:\mathbb{N}}\texttt{Vec}\ A\ n$ is the type of pairs (n, xs) where $n : \mathbb{N}$ and $xs : \texttt{Vec}\ A\ n$.

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

$$\wr|$$

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

Heterogeneous equations

$\Sigma_{n:\mathbb{N}}\texttt{Vec}\ A\ n$ is the type of pairs (n, xs) where $n : \mathbb{N}$ and $xs : \texttt{Vec}\ A\ n$.

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

$$\wr|$$

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

What is the type of e_2?

Heterogeneous equations

Solution: use equation variables as placeholders for their solutions:

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

$$\wr|$$

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

Heterogeneous equations

Solution: use equation variables as placeholders for their solutions:

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

$$\wr|$$

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

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!

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

$$\wr|$$

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

$$\wr|$$

$$\bot$$

Be careful with heterogeneous equations!

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

$$\wr\!|$$

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

$$\not\wr\!|$$

$$\bot$$

The `conflict` rule does not apply!

Be careful with heterogeneous equations!

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

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!

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

$$\wr|$$

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

$$\wr|$$

$$\bot$$

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

Injectivity for indexed data

Do standard unification rules apply to constructors of indexed datatypes?

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

$$\wr|$$

???

Injectivity for indexed data

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

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

Injectivity for indexed data

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

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

Injectivity for indexed data

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

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

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

The image datatype

The type $\texttt{Im}\ f\ y$ consists of elements $\texttt{image}\ x$ such that $f\ x = y$:

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

Solving unsolvable equations

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

Solving unsolvable equations

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

Solving unsolvable equations

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

What if the indices are not fully general?

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

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?

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

$$\wr|$$

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

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

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

$$\wr|$$

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

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

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

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

let's first consider

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

Step 2: solve lower-dimensional equations

This gives us an equivalence f of type

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

Step 3: lift unifier to higher dimension

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

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

Final result of steps 1-3

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

$$\wr|$$

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

Final result of steps 1-3

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

$$\wr|$$

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

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

Lifting equivalences: (mostly) general case

Theorem. If we have an equivalence f of type

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

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

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

Implementation in Agda

This is all used by Agda to check definitions by dependent pattern matching.

- More general than before
- Fixed many bugs
- Implementation matches theory

You can try it for yourself:
`wiki.portal.chalmers.se/agda`

Conclusion

Unification rules should return **evidence** of their correctness.

Conclusion

Unification rules should return **evidence** of their correctness.

A most general unifier can be represented internally as an **equivalence**.

Conclusion

Unification rules should return **evidence** of their correctness.

A most general unifier can be represented internally as an **equivalence**.

Unification cannot ignore the **types**!

Questions?

If you want to know more, you can:

- Try out Agda:
  `wiki.portal.chalmers.se/agda`
- Look at the source:
  `github.com/agda/agda`
- Read my thesis:
  Dependent pattern matching and proof-relevant unification (2017)

Two applications of unification

Filling in implicit arguments

Checking definitions by pattern matching

Two applications of unification

Filling in implicit arguments

- Higher order

Checking definitions by pattern matching

- First order

Two applications of unification

Filling in implicit arguments

- Higher order
- ‘Syntactic’

Checking definitions by pattern matching

- First order
- ‘Semantic’

Two applications of unification

Filling in implicit arguments

- Higher order
- ‘Syntactic’
- MGU optional

Checking definitions by pattern matching

- First order
- ‘Semantic’
- MGU required

Two applications of unification

Filling in implicit arguments

- Higher order
- 'Syntactic'
- MGU optional

Checking definitions by pattern matching

- First order
- 'Semantic'
- MGU required

Focus of this talk

Two notions of equality

Definitional equality

$$x = y : A$$

- Weaker

Propositional equality

$$e : x \equiv_A y$$

- Stronger

Two notions of equality

Definitional equality

$$x = y : A$$

- Weaker
- Decidable

Propositional equality

$$e : x \equiv_A y$$

- Stronger
- Undecidable

Two notions of equality

Definitional equality	Propositional equality
$x = y : A$	$e : x \equiv_A y$

Definitional equality:

- Weaker
- Decidable
- Meta-theoretic

Propositional equality:

- Stronger
- Undecidable
- Internal to theory

Two notions of equality

Definitional equality

$$x = y : A$$

- Weaker
- Decidable
- Meta-theoretic
- Implicit

Propositional equality

$$e : x \equiv_A y$$

- Stronger
- Undecidable
- Internal to theory
- Explicit