Unifiers as Equivalences

Proof-Relevant Unification of Dependently Typed Data

Jesper Cockx
Dominique Devriese
Frank Piessens

20 September 2016

```
data Vec (A : Set) : ℕ → Set where
  []   : 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 xs = { }
```

```
data Vec (A : Set) : ℕ → Set where
  []   : 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 []           = { }  -- suc k = zero
tail k (cons n x xs) = { }  -- suc k = suc n
```

```
data Vec (A : Set) : ℕ → Set where
  []   : 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 (cons .k x xs) = { }
```

```
data Vec (A : Set) : ℕ → Set where
  []   : 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 (cons .k x xs) = xs
```

Introduction

```
data Vec (A : Set) : ℕ → Set where
  []   : 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 (cons .k x xs) = xs
```

- In a dependently typed language, you often encounter equations in the context that you'd like to discharge.
- For example, the indexed datatype `Vec` has two constructors: one for the empty vector of length `zero` and one for prepending an element to an existing vector, increasing the length by 1. When you want to implement a type-safe tail function on vectors, you have to do a case analysis on a vector of length `suc k`, resulting in the two equations `suc k = zero` and `suc k = suc n`.
- Agda automatically detects that the first case is impossible and that $k = n$ in the second case. How does it do this?

Agda uses unification to:

- eliminate impossible cases
- specialize the result type

Agda uses unification to:

- eliminate impossible cases
- specialize the result type

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

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?

Introduction

Agda uses unification to:

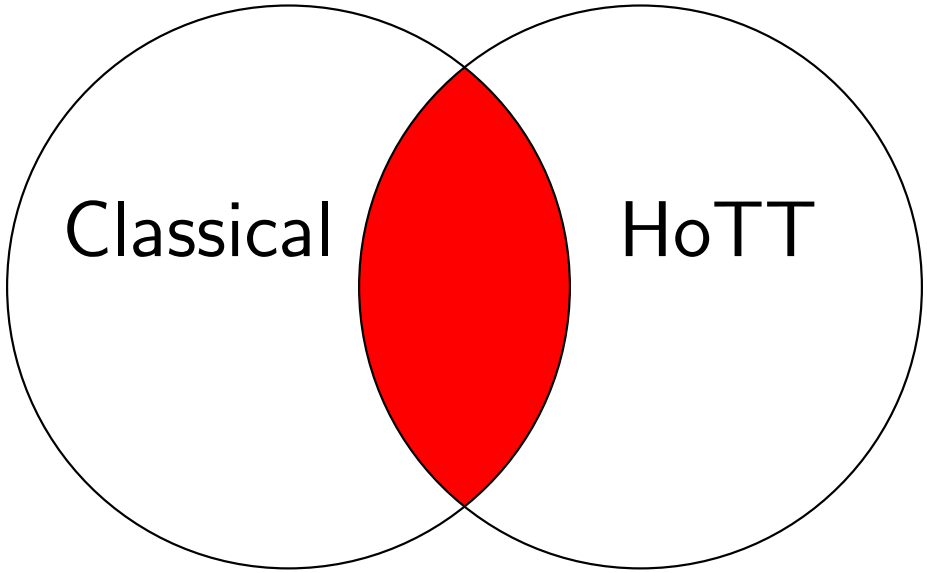
- eliminate impossible cases
- specialize the result type

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

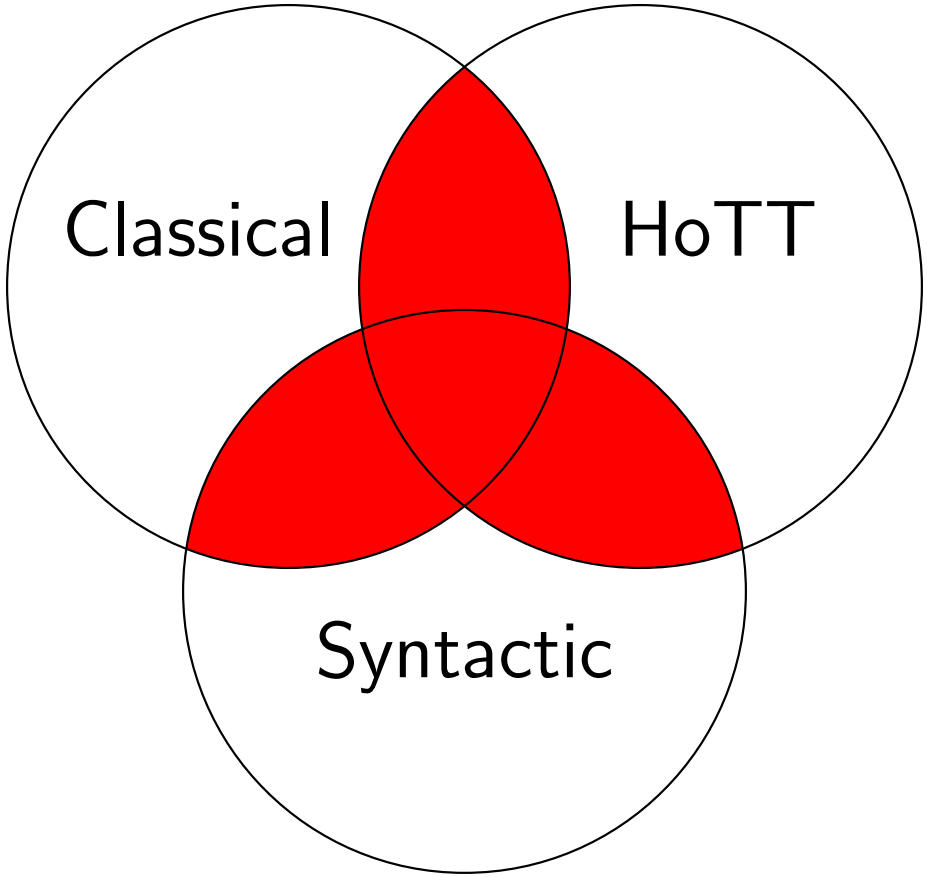
Main question: How to make sure the output of unification is correct?

- The answer is in the title: Agda applies *unification* to solve these equations automatically.
- Similar equations arise in other dependently typed languages, e.g. in Coq you may use constructors with embedded equality proofs instead of an indexed datatype. So unification can also be applied there.
- The main question I will try to answer in this presentation is: how can we be sure the output of unification is correct?
- In particular, I argue that the naieve idea of unification as finding a substitution making two terms equal is *not sufficient*.

Flavors of type theory

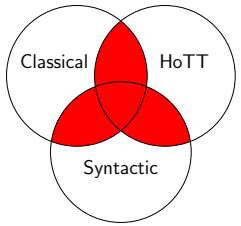


Flavors of type theory



Introduction

Flavors of type theory



- Let's start with the question why the standard definition of a most general unifier isn't sufficient.
- For this, we first need to zoom out. Intuitionistic type theory can be seen as a vanilla theory plus a number of flavors in the form of axioms or new primitives.
- For example, you can add a classical flavor such as the law of the excluded middle, impredicativity, and uniqueness of identity proofs.
- On the other hand, you can add homotopy flavor with primitives such as functional extensionality, univalence, and higher inductive types.
- However, using these flavors together blows up the whole theory, making it inconsistent.
- There's a third flavor that I'd call the syntactic properties. These are the properties that are true in a syntactic model.
- For example, there's *injectivity of type constructors*, stating that e.g. $\texttt{List}\ A = \texttt{List}\ B$ implies $A = B$.
- These properties are in general incompatible with both classical logic and HoTT, so we want to avoid them if possible.
- However, a purely syntactic unification algorithm implicitely relies on these properties to justify its steps.
- To make sure the output of unification is consistent with whatever flavor we're working in, we need evidence of unification internal to our theory.

We want something that works for *all* flavors, so a purely syntactic algorithm doesn't work.

We want something that works for *all* flavors, so a purely syntactic algorithm doesn't work.

Core idea: unification should return *evidence* of unification in the form of an *equivalence*

$$(a \equiv b) \simeq (c \equiv d)$$

Introduction

We want something that works for *all* flavors, so a purely syntactic algorithm doesn't work.

Core idea: unification should return *evidence* of unification in the form of an *equivalence*

$$(a \equiv b) \simeq (c \equiv d)$$

- My answer to this problem is that you should think of unifiers as type-theoretic *equivalences* between two equations. An equivalence means (roughly) that we have functions back and forth that are mutually inverses.
- This means we give a computational interpretation to the concept of a unifier: not just a substitution, but functions manipulating identity proofs.
- By requiring evidence of unification internal to the type theory, we make sure the unification doesn't rely on any unspecified assumptions (e.g. uniqueness of identity proofs or injective type constructors).
- Additionally, it can be used in the translation of dependent pattern matching to eliminators

Unifiers as equivalences

Proof-relevant unification

Depending on equations

Introduction

- First I'll explain why it's a good idea to see unifiers as equivalences
- Next I'll show concretely how the standard unification rules can be viewed as equivalences
- Finally I'll go more into what happens when dependently typed terms themselves become the subject of unification

Unifiers as equivalences

Proof-relevant unification

Depending on equations

What is a unification problem?

A *unification problem* consists of

1. A context of free variables Γ
2. Equations $u_1 = v_1$, $u_2 = v_2$, ...

Unification problems are telescopes!

A *unification problem* consists of

1. A context of free variables Γ
2. Equations $u_1 = v_1$, $u_2 = v_2$, ...

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 equivalences

What is a unification problem?

- So, to begin we need to think about what a unification problem is. We know that it should consist of one or more equations and that these equations can contain free variables that we are trying to solve.
- Of course, we take a typed view on unification, so we collect the unification variables in a context assigning a type to each variable.
- For the internal representation of the equations, we make use of Martin-Löf's identity type. This type is written with a triple equals sign in Agda, I will be using this notation as well.
- The bar above u and v simply means that there may be more than one equation.
- For easy reference, we also give each equation a name ($\bar{e}$ in this case). This will become important once we discuss dependencies between equations in the third part of the presentation.

Unification problems are telescopes!

A *unification problem* consists of

1. A context of free variables Γ
2. Equations $u_1 = v_1$, $u_2 = v_2$, ...

This can be represented as a *telescope*

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

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

What is a unifier?

A *unifier* of $\bar{u}$ and $\bar{v}$ consists of:

1. A reduced context Γ'
2. A substitution $\sigma : \Gamma' \to \Gamma$ s.t. $\bar{u}\sigma = \bar{v}\sigma$

Unifiers are telescope maps!

A *unifier* of $\bar{u}$ and $\bar{v}$ consists of:

1. A reduced context Γ'
2. A substitution $\sigma : \Gamma' \to \Gamma$ s.t. $\bar{u}\sigma = \bar{v}\sigma$

This can be represented as a *telescope map*

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

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

Unifiers as equivalences

What is a unifier?

- A unifier is usually defined as any substitution σ that makes all the equations true. Since we take a typed view on unification, we also make the domain of the substitution, Γ', explicit. Note that Γ' contains the variables that are *not* assigned a value by σ.
- We can encode both the substitution σ and the fact that it makes the equations hold together as a *telescope map*. This is simply a function that takes its arguments from Γ' and returns the values of the variables in Γ plus *proofs* that the equations hold under this substitution.
- For example, if we had one variable n and one equation $n = \texttt{zero}$ then Γ' is empty and f assigns `zero` to n and `refl` to e.

Unifiers are telescope maps!

A *unifier* of $\bar{u}$ and $\bar{v}$ consists of:

1. A reduced context Γ'
2. A substitution $\sigma : \Gamma' \to \Gamma$ s.t. $\bar{u}\sigma = \bar{v}\sigma$

This can be represented as a *telescope map*

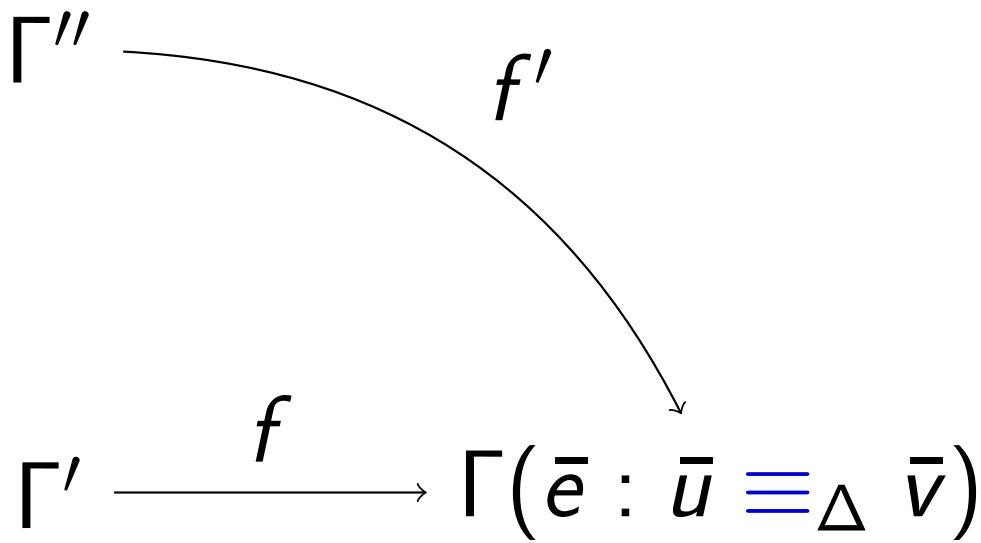
$$f : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_A \bar{v})$$

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

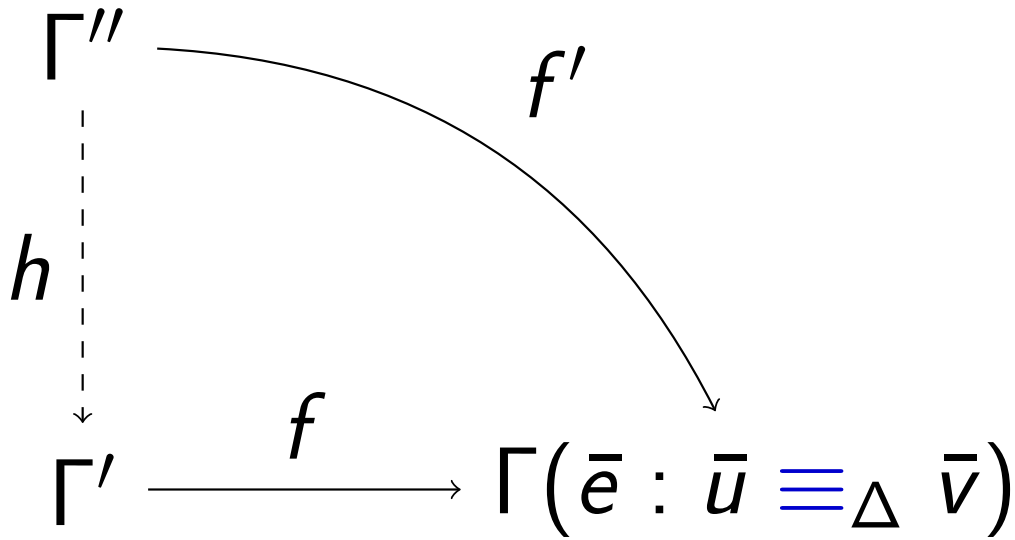
What is a most general unifier?

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

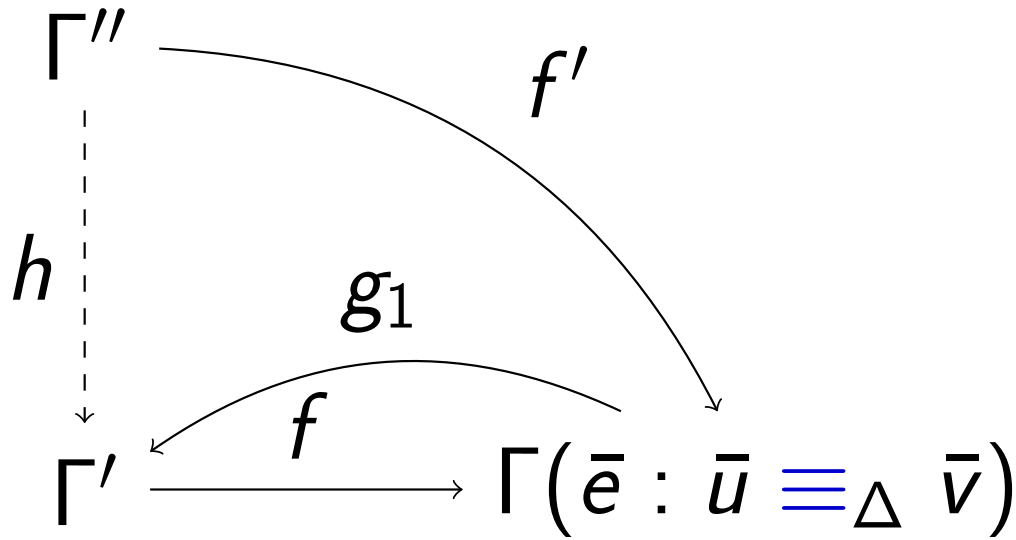
What is a most general unifier?



What is a most general unifier?

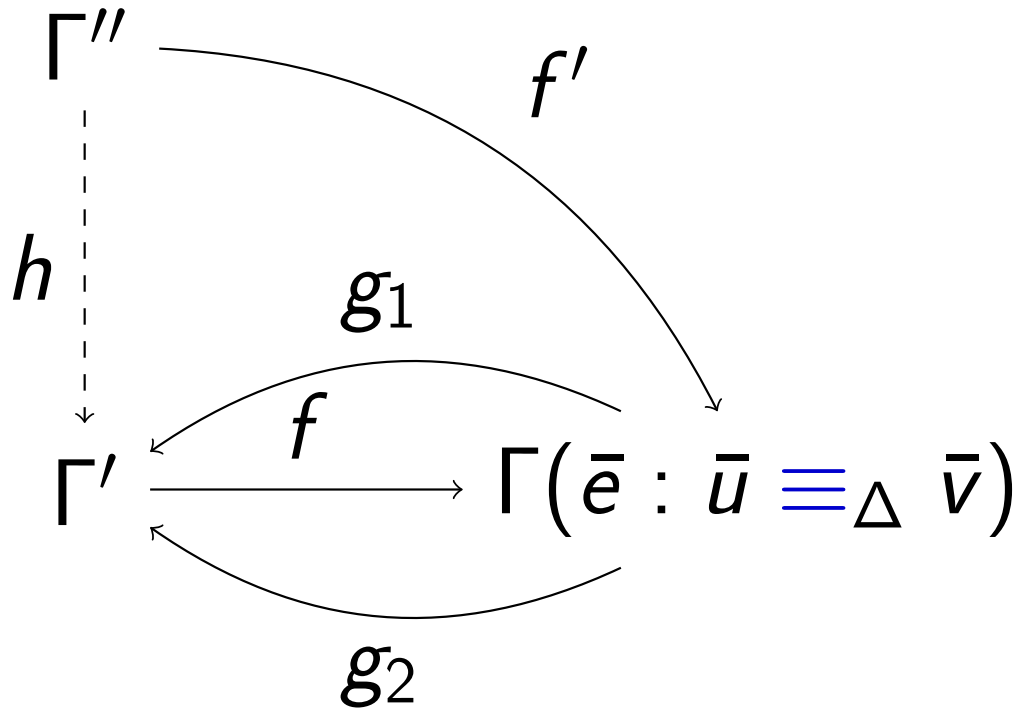


What is a most general unifier?



f has a *right inverse* $g_1 \Rightarrow h$ exists

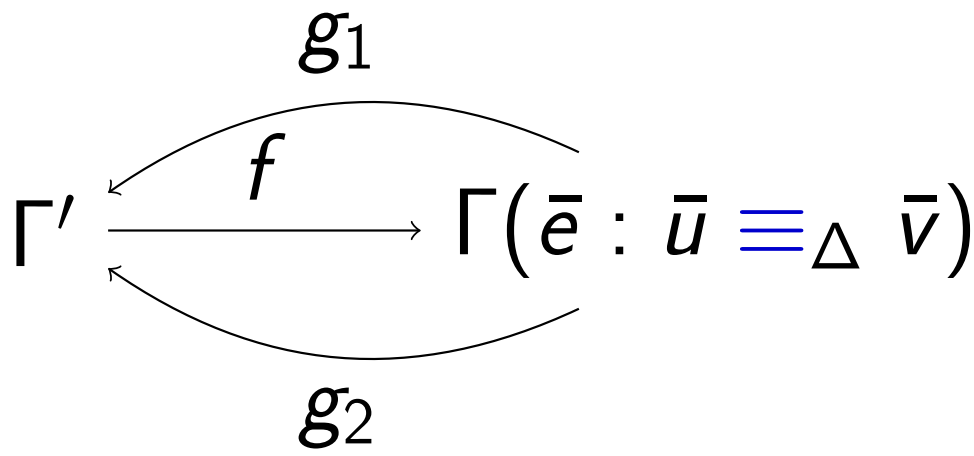
What is a most general unifier?



f has a *right inverse* $g_1 \Rightarrow h$ exists

f has a *left inverse* $g_2 \Rightarrow h$ is unique

What is a most general unifier?

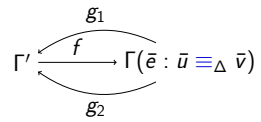


f has a *right inverse* g_1

f has a *left inverse* g_2

Unifiers as equivalences

What is a most general unifier?



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

- We call a unifier $f : \Gamma' \to \Gamma(\bar{e} : \bar{u} \equiv_A \bar{v})$ most general if any other unifier $f' : \Gamma'' \to \Gamma(\bar{e} : \bar{u} \equiv_A \bar{v})$ can be decomposed as $f \circ h$.
- However, this definition quantifies over all telescopes Γ'' and unifiers f', which is annoying. Can we find a better definition?
- If we require that f has a *right inverse* g_1, we don't need h any more, since we can always define it as $g_1 \circ f'$!
- (If anyone asks how to construct g_1: Take $\Gamma'' = \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v})$ and $f' = \texttt{id}$. This gives us a function $g_1 : \Gamma(\bar{e} : \bar{u} \equiv_\Delta \bar{v}) \to \Gamma'$ such that $\texttt{id} = f \circ g_1$, i.e. g_1 is a right inverse to f.)
- Usually it is also required that the substitution h is unique, otherwise Γ' may contain unneccessary 'ghost variables'.
- If we require that f also has a *left inverse* g_2, we don't need uniqueness of h either.
- Note that g_1 and g_2 don't have to be the same, but they can be (and often are).
- (If anyone asks how to construct g_2: Note that we have two functions h from Γ' to Γ' such that $f \circ h = f$: $h = g_1 \circ f$ and $h = \texttt{id}$. By uniqueness, we must have $g_1 \circ f = \texttt{id}$, so g_1 is also a left inverse to f.)

Most general unifiers are equivalences!

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

Most general unifiers are equivalences!

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

- So now we have a function f with left and a right inverses. In type-theoretic circles, this is called an *equivalence*, famous for its role in Voevodsky's univalence axiom.
- This is great because there already is a great amount of theory dealing with equivalences that we can borrow.

Unifiers as equivalences

Proof-relevant unification

Depending on equations

Example

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

Example

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

$$\wr$$

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

Example

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

$$\wr$$

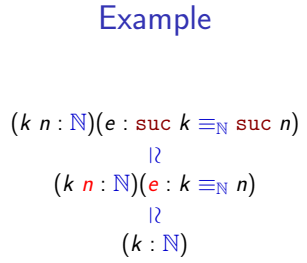
$$(k\ n : \mathbb{N})(e : k \equiv_{\mathbb{N}} n)$$

$$\wr$$

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

Proof-relevant unification

Example



- Now that we know what a most general unifier is, we can try to construct them. We start with an easy example from the introduction: $\mathtt{suc}\ k = \mathtt{suc}\ n$.
- We construct the most general unifier by applying unification rules that simplify the equations. Each unification rule also takes the form of an equivalence.
- These equivalences can be chained together by transitivity of the equivalence relation, thus producing the final MGU in the end.

The solution rule

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

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

- The most basic unification rule is the solution rule. It takes a variable and an equation having this variable on one side and solves it. This removes the variable in the process.
- On the right of the equivalence is the empty telescope. You can think of it as a unit type with a single element.
- The function from right to left in the equivalence assigns t to the variable x and `refl` to e.

The deletion rule

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

The deletion rule

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

Requires uniqueness of identity proofs!

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

Requires uniqueness of identity proofs!

- The next basic unification rule is the deletion rule. It removes a reflexive equation from the telescope, leaving the rest of it unchanged.
- The construction of `deletion` requires uniqueness of identity proofs however, so think before including this rule in your unification algorithm!

The injectivity rule

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

Proof-relevant unification

The injectivity rule

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

- Next up is the injectivity rule: if we have an equation between two equal constructors, we can simplify it to an equation between the arguments.
- It's important that the constructors are fully applied, otherwise we may run into trouble with functional extensionality!

The conflict rule

$$\texttt{conflict}_{\texttt{left},\texttt{right}} : (e : \texttt{left}\ x \equiv_{A \uplus B} \texttt{right}\ y) \simeq \bot$$

$\texttt{conflict}_{\texttt{left},\texttt{right}} : (e : \texttt{left}\ x \equiv_{A \uplus B} \texttt{right}\ y) \simeq \bot$

- Next to the basic unification rules we just saw, there are also rules for detecting absurd equations. In the spirit of this talk, we represent also these rules as equivalences, but this time with the empty type $\bot$ on the right.
- The conflict rule can be applied when there is an equation between two distinct constructors. Again, both constructors should be fully applied.
- Since the right side is $\bot$, the only interesting information in this equivalence is the function from left to right.

The cycle rule

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

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

- Finally, there is the cycle rule. This rule can be applied when the term on the left occurs strongly rigid on the right, i.e. as a (nested) constructor argument.

Unifiers as equivalences

Proof-relevant unification

Depending on equations

What's the type of a heterogeneous equation?

$$(e : \mathbb{N}, \texttt{zero} \equiv_{\Sigma_{A:\texttt{Set}}\, A} \texttt{Bool}, \texttt{true})$$

What's the type of a heterogeneous equation?

$$(e : \mathbb{N}, \mathtt{zero} \equiv_{\Sigma_{A:\mathtt{Set}} A} \mathtt{Bool}, \mathtt{true})$$

$$\wr$$

$$(e_1 : \mathbb{N} \equiv_{\mathtt{Set}} \mathtt{Bool})(e_2 : \mathtt{zero} \equiv_{???} \mathtt{true})$$

Depending on equations

What's the type of a heterogeneous equation?

$$(e : \mathbb{N}, \texttt{zero} \equiv_{\Sigma_{A:\texttt{Set}} A} \texttt{Bool}, \texttt{true})$$
$$\wr$$
$$(e_1 : \mathbb{N} \equiv_{\texttt{Set}} \texttt{Bool})(e_2 : \texttt{zero} \equiv_{???} \texttt{true})$$

- When we try to unify dependently typed terms, we can encounter heterogeneous equations: equations where the left- and right-hand side don't have the same type. For example, we may have an equation between pairs of a type and an element of that type.
- Can we allow heterogeneous equalities? If yes, can we still apply the standard unification rules to them?

Why not use heterogeneous equality?

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

vs

$$(e : \texttt{Bool}, \texttt{true} \equiv_{\texttt{Set} \times \texttt{Bool}} \texttt{Bool}, \texttt{false})$$

Why not use heterogeneous equality?

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

vs

$(e : \texttt{Bool}, \texttt{true} \equiv_{\texttt{Set} \times \texttt{Bool}} \texttt{Bool}, \texttt{false})$

- To answer this question, consider the following two unification problems. They look very much alike, except that the type of the first one is a dependent product $\Sigma_{A:\texttt{Set}} A$ while the second one has a non-dependent product $\texttt{Set} \times \texttt{Bool}$ as its type.
- If we used heterogeneous equality, both equations would be simplified to the same two equations $\texttt{Bool} = \texttt{Bool}$ and $\texttt{true} = \texttt{false}$.
- However, the first equation is actually provable if you use the univalence axiom, while the second one is false in any type theory. So heterogeneous equality loses information that is essential to the problem!

Telescopic equality

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

$$(E : \mathbb{N} \equiv_{\mathtt{Set}} \mathtt{Bool})(e : \mathtt{zero} \equiv_E \mathtt{true})$$

Telescopic equality

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

$$(E : \mathbb{N} \equiv_{\mathtt{Set}} \mathtt{Bool})(e : \mathtt{zero} \equiv_E \mathtt{true})$$

This is called a *telescopic equality*

Depending on equations

Telescopic equality

- Instead, we solve the problem by using *telescopic equality*. This means that the name of each equation can occur in the types of subsequent equations.
- This means we can keep track precisely how the type of each equation depends on the previous equations, and in particular when it becomes again homogeneous. If an equation is homogeneous, we know it's safe to apply the unification rules to it.
- Telescopic equalities can be formalized by using the 'path over a path' construction from homotopy type theory. Our notation in particular is inspired by cubical type theory.

Telescopic equality

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

$$(E : \mathbb{N} \equiv_{\mathsf{Set}} \mathtt{Bool})(e : \mathtt{zero} \equiv_{E} \mathtt{true})$$

This is called a *telescopic equality*

Exploiting the dependencies between equations

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

Exploiting the dependencies between equations

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

Depending on equations

Exploiting the dependencies

Exploiting the dependencies between equations

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

- Telescopic equalities don't just tell us when it's safe to apply unification rules. They also play an essential role in the unification rules for *indexed datatypes*.
- For example, consider the two equations e_1 and e_2, where the type of the second one depends on the first one. The injectivity rule for the constructor `cons` of the `Vec` datatype takes both of these equations at once and simplifies them to equations between the constructor arguments.
- In general, the unification rules for indexed datatypes always solve the equations between the indices together with equations between constructors themselves.
- The reason why the rules work in this way is because it is the *total* type $\Sigma_{(n:\mathbb{N})}\ \texttt{Vec}\ A\ n$ of an indexed datatype that is inductively defined, not the individual types $\texttt{Vec}\ A\ n$.

Solving unsolvable equations

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

Solving unsolvable equations

$$\textbf{data}\ \mathtt{Im}\ (f : A \to B) : B \to \mathtt{Set}\ \textbf{where}$$
$$\mathtt{image} : (x : A) \to \mathtt{Im}\ f\ (f\ x)$$

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

Solving unsolvable equations

$$\textbf{data}\ \mathtt{Im}\ (f : A \to B) : B \to \mathtt{Set}\ \textbf{where}$$
$$\mathtt{image} : (x : A) \to \mathtt{Im}\ f\ (f\ x)$$

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

Solving unsolvable equations

$$\textbf{data}\ \mathtt{Im}\ (f : A \to B) : B \to \mathtt{Set}\ \textbf{where}$$
$$\mathtt{image} : (x : A) \to \mathtt{Im}\ f\ (f\ x)$$

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

Solving unsolvable equations

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

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

- Some people consider this datatype to be criminal. I ask these people kindly to imagine the `image` constructor takes an additional argument of type $f\ x \equiv_B y$.
- You can think of $\texttt{Im}\ f\ y$ as the set of $x : A$ such that $f\ x = y$.
- If we apply the injectivity rule to the `image` constructor, we see that it simplifies the two equations $f\ x \equiv_B f\ y$ and $\texttt{image}\ x \equiv_{\texttt{Im}\ f\ e_1} \texttt{image}\ x_2$ to the single equation $x_1 \equiv_A x_2$.
- This is neat because it would've been impossible to solve the equation $f\ x = f\ y$ by itself. Hooray for the power of dependent types!

Things I didn't mention

- Construction of the unification rules

Things I didn't mention

- Construction of the unification rules
- Computational interpretation of unifiers

Things I didn't mention

- Construction of the unification rules
- Computational interpretation of unifiers
- Eta rules for record types

Things I didn't mention

- Construction of the unification rules
- Computational interpretation of unifiers
- Eta rules for record types
- Reverse unification rules (outdated)

Things I didn't mention

- Construction of the unification rules
- Computational interpretation of unifiers
- Eta rules for record types
- Reverse unification rules (outdated)
- Implementation in Agda

Conclusion

Things I didn't mention

- Read the paper if you want to know about these things!
- I'm also working on an extension to the algorithm called *higher-dimensional unification* that replaces the reverse unification rules in the paper. You'll hear more about that in the future.

Conclusion

We have a new definition of the MGU

... internal to the type theory

Conclusion

We have a new definition of the MGU

. . . internal to the type theory

. . . that is correct by construction

Conclusion

We have a new definition of the MGU

- ... internal to the type theory
- ... that is correct by construction
- ... and can be used to compile pattern matching to eliminators

Conclusion

Conclusion

Conclusion

We have a new definition of the MGU

... internal to the type theory

... that is correct by construction

... and can be used to compile pattern matching to eliminators

- In a dependently typed language, it is possible to enforce correctness properties internal to the language. We apply this idea to unification, discovering that unifiers can be represented internally as equivalences.
- This idea allows us to give a new implementation of the unification algorithm used by Agda for dependent pattern matching, that avoids many of the problems troubling the old algorithm.
- Additionally, by giving a computational interpretation to unifiers we can use them directly in our type-theoretic developments, for example in the translation of dependent pattern matching to eliminators.