



DATA
61

COMP4161: Advanced Topics in Software Verification



Gerwin Klein, Johannes Åman Pohjola, Christine Rizkallah, Miki Tanaka
T3/2020

data61.csiro.au



Last time...



- λ calculus syntax
- free variables, substitution
- β reduction
- α and η conversion
- β reduction is confluent
- λ calculus is expressive (Turing complete)
- λ calculus is inconsistent (as a logic)

Content



→ Foundations & Principles

- Intro, Lambda calculus, natural deduction [1,2]
- Higher Order Logic, Isar (part 1) [2,3^a]
- Term rewriting [3,4]

→ Proof & Specification Techniques

- Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
- General recursive functions, termination proofs [7^b]
- Proof automation, Hoare logic, proofs about programs, invariants [8]
- C verification [9,10]
- Practice, questions, exam prep [10^c]

^aa1 due; ^ba2 due; ^ca3 due

λ calculus is inconsistent



Can find term R such that $R R =_{\beta} \text{not}(R R)$

There are more terms that do not make sense:

$1\ 2$, true false , etc.

λ calculus is inconsistent



Can find term R such that $R R =_{\beta} \text{not}(R R)$

There are more terms that do not make sense:

$1\ 2$, true false , etc.

Solution: rule out ill-formed terms by using types.
(Church 1940)

Introducing types



Idea: assign a type to each “sensible” λ term.

Examples:

Introducing types



Idea: assign a type to each “sensible” λ term.

Examples:

→ for *term* t has type α write $t :: \alpha$

Introducing types



Idea: assign a type to each “sensible” λ term.

Examples:

- for *term* t has type α write $t :: \alpha$
- if x has type α then $\lambda x. x$ is a function from α to α
Write: $(\lambda x. x) :: \alpha \Rightarrow \alpha$

Introducing types



Idea: assign a type to each “sensible” λ term.

Examples:

- for *term* t has type α write $t :: \alpha$
- if x has type α then $\lambda x. x$ is a function from α to α
Write: $(\lambda x. x) :: \alpha \Rightarrow \alpha$
- for $s t$ to be sensible:
 s must be a function
 t must be right type for parameter
If $s :: \alpha \Rightarrow \beta$ and $t :: \alpha$ then $(s t) :: \beta$



DATA
61



That's about it



DATA
61



Now formally again

Syntax for λ^{\rightarrow}



Terms: $t ::= v \mid c \mid (t t) \mid (\lambda x. t)$
 $v, x \in V, \quad c \in C, \quad V, C$ sets of names

Types: $\tau ::= b \mid \nu \mid \tau \Rightarrow \tau$
 $b \in \{\text{bool}, \text{int}, \dots\}$ base types
 $\nu \in \{\alpha, \beta, \dots\}$ type variables

$$\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

Syntax for λ^{\rightarrow}



Terms: $t ::= v \mid c \mid (t t) \mid (\lambda x. t)$
 $v, x \in V, \quad c \in C, \quad V, C$ sets of names

Types: $\tau ::= b \mid \nu \mid \tau \Rightarrow \tau$
 $b \in \{\text{bool}, \text{int}, \dots\}$ base types
 $\nu \in \{\alpha, \beta, \dots\}$ type variables

$$\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

Context Γ :

Γ : function from variable and constant names to types.

Syntax for λ^{\rightarrow}



Terms: $t ::= v \mid c \mid (t t) \mid (\lambda x. t)$
 $v, x \in V, \quad c \in C, \quad V, C$ sets of names

Types: $\tau ::= b \mid \nu \mid \tau \Rightarrow \tau$
 $b \in \{\text{bool}, \text{int}, \dots\}$ base types
 $\nu \in \{\alpha, \beta, \dots\}$ type variables

$$\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

Context Γ :

Γ : function from variable and constant names to types.

Term t has type τ in context Γ : $\Gamma \vdash t :: \tau$

Examples



$$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$$

Examples


$$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$$
$$[y \leftarrow \text{int}] \vdash y :: \text{int}$$

Examples


$$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$$
$$[y \leftarrow \text{int}] \vdash y :: \text{int}$$
$$[z \leftarrow \text{bool}] \vdash (\lambda y. y) z :: \text{bool}$$

Examples


$$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$$
$$[y \leftarrow \text{int}] \vdash y :: \text{int}$$
$$[z \leftarrow \text{bool}] \vdash (\lambda y. y) z :: \text{bool}$$
$$[] \vdash \lambda f x. f x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$$

Examples


$$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$$
$$[y \leftarrow \text{int}] \vdash y :: \text{int}$$
$$[z \leftarrow \text{bool}] \vdash (\lambda y. y) z :: \text{bool}$$
$$[] \vdash \lambda f x. f x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$$

A term t is **well typed** or **type correct**
if there are Γ and τ such that $\Gamma \vdash t :: \tau$

Type Checking Rules



Variables:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)}$$

Type Checking Rules



Variables: $\frac{}{\Gamma \vdash x :: \Gamma(x)}$

Application: $\frac{}{\Gamma \vdash (t_1 t_2) :: \tau}$

Type Checking Rules



Variables: $\overline{\Gamma \vdash x :: \Gamma(x)}$

Application: $\frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1 t_2) :: \tau}$

Type Checking Rules



Variables: $\overline{\Gamma \vdash x :: \Gamma(x)}$

Application: $\frac{\Gamma \vdash t_1 :: \tau_1 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1 t_2) :: \tau}$

Abstraction: $\overline{\Gamma \vdash (\lambda x. t) :: \tau_x \Rightarrow \tau}$

Type Checking Rules



Variables: $\overline{\Gamma \vdash x :: \Gamma(x)}$

Application: $\frac{\Gamma \vdash t_1 :: \tau_1 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1 t_2) :: \tau}$

Abstraction: $\frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \tau_x \Rightarrow \tau}$

Example Type Derivation:



Example Type Derivation:



$$\frac{}{\Box \vdash \lambda x y. x :: \alpha \Rightarrow \beta \Rightarrow \alpha}$$

Example Type Derivation:



$$\frac{[x \leftarrow \alpha] \vdash \lambda y. x :: \beta \Rightarrow \alpha}{\square \vdash \lambda x y. x :: \alpha \Rightarrow \beta \Rightarrow \alpha}$$

Example Type Derivation:



$$\frac{\frac{\frac{}{[x \leftarrow \alpha, y \leftarrow \beta] \vdash x :: \alpha}}{[x \leftarrow \alpha] \vdash \lambda y. x :: \beta \Rightarrow \alpha}}{\square \vdash \lambda x y. x :: \alpha \Rightarrow \beta \Rightarrow \alpha}}$$

More complex Example



$$\square \vdash \lambda f x. f x x ::$$

More complex Example



$$\overline{\square \vdash \lambda f x. f x x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

More complex Example



$$\frac{\overline{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f x x :: \alpha \Rightarrow \beta}}{\boxed{\vdash \lambda f x. f x x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}}$$

More complex Example



$$\frac{\frac{\Gamma \vdash f \ x \ x :: \beta}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f \ x \ x :: \alpha \Rightarrow \beta}}{[] \vdash \lambda f \ x. f \ x \ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$

More complex Example



$$\frac{\frac{\frac{\Gamma \vdash f x :: \alpha \Rightarrow \beta}{\Gamma \vdash f x x :: \beta}}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f x x :: \alpha \Rightarrow \beta}}{[] \vdash \lambda f x. f x x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$

More complex Example



$$\frac{\frac{\frac{\Gamma \vdash f x :: \alpha \Rightarrow \beta}{\Gamma \vdash f x x :: \beta}}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f x x :: \alpha \Rightarrow \beta}}{[] \vdash \lambda f x. f x x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$

More complex Example



$$\frac{\frac{\frac{\overline{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)}}{\Gamma \vdash f x :: \alpha \Rightarrow \beta} \quad \overline{\Gamma \vdash x :: \alpha}}{\Gamma \vdash f x x :: \beta}}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f x x :: \alpha \Rightarrow \beta}}{\boxed{\vdash} \lambda f x. f x x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$

More complex Example



$$\frac{\frac{\frac{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)}{\Gamma \vdash f x :: \alpha \Rightarrow \beta} \quad \frac{\Gamma \vdash x :: \alpha}{\Gamma \vdash x :: \alpha}}{\Gamma \vdash f x x :: \beta}}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f x x :: \alpha \Rightarrow \beta}}{\boxed{\vdash} \lambda f x. f x x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$

More general Types



A term can have more than one type.

More general Types



A term can have more than one type.

Example: $\square \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$
 $\square \vdash \lambda x. x :: \alpha \Rightarrow \alpha$

More general Types



A term can have more than one type.

Example: $\square \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$
 $\square \vdash \lambda x. x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

$\tau \lesssim \sigma$ if there is a substitution S such that $\tau = S(\sigma)$

More general Types



A term can have more than one type.

Example: $\square \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$
 $\square \vdash \lambda x. x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

$\tau \lesssim \sigma$ if there is a substitution S such that $\tau = S(\sigma)$

Examples:

$\text{int} \Rightarrow \text{bool} \lesssim \alpha \Rightarrow \beta$

More general Types



A term can have more than one type.

Example: $\square \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$
 $\square \vdash \lambda x. x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

$\tau \lesssim \sigma$ if there is a substitution S such that $\tau = S(\sigma)$

Examples:

$\text{int} \Rightarrow \text{bool} \lesssim \alpha \Rightarrow \beta \lesssim \beta \Rightarrow \alpha$

More general Types



A term can have more than one type.

Example: $\square \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$
 $\square \vdash \lambda x. x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

$\tau \lesssim \sigma$ if there is a substitution S such that $\tau = S(\sigma)$

Examples:

$\text{int} \Rightarrow \text{bool} \lesssim \alpha \Rightarrow \beta \lesssim \beta \Rightarrow \alpha \not\lesssim \alpha \Rightarrow \alpha$

Most general Types



Fact: each type correct term has a most general type

Most general Types



Fact: each type correct term has a most general type

Formally:

$$\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \wedge (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \lesssim \sigma)$$

Most general Types



Fact: each type correct term has a most general type

Formally:

$$\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \wedge (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \lesssim \sigma)$$

It can be found by executing the typing rules backwards.

Most general Types



Fact: each type correct term has a most general type

Formally:

$$\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \wedge (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \lesssim \sigma)$$

It can be found by executing the typing rules backwards.

→ **type checking:** checking if $\Gamma \vdash t :: \tau$ for given Γ and τ

Most general Types



Fact: each type correct term has a most general type

Formally:

$$\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \wedge (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \lesssim \sigma)$$

It can be found by executing the typing rules backwards.

- **type checking:** checking if $\Gamma \vdash t :: \tau$ for given Γ and τ
- **type inference:** computing Γ and τ such that $\Gamma \vdash t :: \tau$

Most general Types



Fact: each type correct term has a most general type

Formally:

$$\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \wedge (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \lesssim \sigma)$$

It can be found by executing the typing rules backwards.

- **type checking:** checking if $\Gamma \vdash t :: \tau$ for given Γ and τ
- **type inference:** computing Γ and τ such that $\Gamma \vdash t :: \tau$

Type checking and type inference on λ^{\rightarrow} are decidable.

What about β reduction?



What about β reduction?



Definition of β reduction stays the same.

What about β reduction?



Definition of β reduction stays the same.

Fact: Well typed terms stay well typed during β reduction

Formally: $\Gamma \vdash s :: \tau \wedge s \rightarrow_{\beta} t \implies \Gamma \vdash t :: \tau$

What about β reduction?



Definition of β reduction stays the same.

Fact: Well typed terms stay well typed during β reduction

Formally: $\Gamma \vdash s :: \tau \wedge s \rightarrow_{\beta} t \implies \Gamma \vdash t :: \tau$

This property is called **subject reduction**

What about termination?



What about termination?



β reduction in $\lambda \rightarrow$ always terminates.



(Alan Turing, 1942)

What about termination?



β reduction in $\lambda \rightarrow$ always terminates.



(Alan Turing, 1942)

→ $=_{\beta}$ is decidable

To decide if $s =_{\beta} t$, reduce s and t to normal form (always exists, because \rightarrow_{β} terminates), and compare result.

What about termination?

β reduction in $\lambda \rightarrow$ always terminates.



(Alan Turing, 1942)

→ $=_{\beta}$ is decidable

To decide if $s =_{\beta} t$, reduce s and t to normal form (always exists, because \rightarrow_{β} terminates), and compare result.

→ $=_{\alpha\beta\eta}$ is decidable

This is why Isabelle can automatically reduce each term to $\beta\eta$ normal form.

What does this mean for Expressiveness?



What does this mean for Expressiveness?



Not all computable functions can be expressed in $\lambda \rightarrow$!

What does this mean for Expressiveness?



Not all computable functions can be expressed in λ^{\rightarrow} !

How can typed functional languages then be turing complete?

What does this mean for Expressiveness?



Not all computable functions can be expressed in λ^{\rightarrow} !

How can typed functional languages then be turing complete?

Fact:

Each computable function can be encoded as closed, type correct λ^{\rightarrow} term using $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$ with $Y t \longrightarrow_{\beta} t (Y t)$ as only constant.

What does this mean for Expressiveness?



Not all computable functions can be expressed in λ^{\rightarrow} !

How can typed functional languages then be turing complete?

Fact:

Each computable function can be encoded as closed, type correct λ^{\rightarrow} term using $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$ with $Y t \longrightarrow_{\beta} t (Y t)$ as only constant.

- Y is called fix point operator
- used for recursion
- lose decidability (what does $Y (\lambda x. x)$ reduce to?)
- (Isabelle/HOL doesn't have Y ; it supports more restricted forms of recursion)

Types and Terms in Isabelle



Types: $\tau ::= b \mid 'v \mid 'v :: C \mid \tau \Rightarrow \tau \mid (\tau, \dots, \tau) K$

$b \in \{\text{bool}, \text{int}, \dots\}$ base types

$v \in \{\alpha, \beta, \dots\}$ type variables

$K \in \{\text{set}, \text{list}, \dots\}$ type constructors

$C \in \{\text{order}, \text{linord}, \dots\}$ type classes

Terms: $t ::= v \mid c \mid ?v \mid (t t) \mid (\lambda x. t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

Types and Terms in Isabelle



Types: $\tau ::= b \mid \nu \mid \nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \dots, \tau) K$

$b \in \{\text{bool}, \text{int}, \dots\}$ base types

$\nu \in \{\alpha, \beta, \dots\}$ type variables

$K \in \{\text{set}, \text{list}, \dots\}$ type constructors

$C \in \{\text{order}, \text{linord}, \dots\}$ type classes

Terms: $t ::= v \mid c \mid ?v \mid (t t) \mid (\lambda x. t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

→ **type constructors:** construct a new type out of a parameter type.

Example: `int list`

Types and Terms in Isabelle



Types: $\tau ::= b \mid \nu \mid \nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \dots, \tau) K$

$b \in \{\text{bool}, \text{int}, \dots\}$ base types

$\nu \in \{\alpha, \beta, \dots\}$ type variables

$K \in \{\text{set}, \text{list}, \dots\}$ type constructors

$C \in \{\text{order}, \text{linord}, \dots\}$ type classes

Terms: $t ::= v \mid c \mid ?v \mid (t t) \mid (\lambda x. t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

- **type constructors:** construct a new type out of a parameter type.
Example: `int list`
- **type classes:** restrict type variables to a class defined by axioms.
Example: `$\alpha :: \text{order}$`

Types and Terms in Isabelle



Types: $\tau ::= b \mid \nu \mid \nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \dots, \tau) K$

$b \in \{\text{bool}, \text{int}, \dots\}$ base types

$\nu \in \{\alpha, \beta, \dots\}$ type variables

$K \in \{\text{set}, \text{list}, \dots\}$ type constructors

$C \in \{\text{order}, \text{linord}, \dots\}$ type classes

Terms: $t ::= v \mid c \mid ?v \mid (t t) \mid (\lambda x. t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

- **type constructors:** construct a new type out of a parameter type.
Example: `int list`
- **type classes:** restrict type variables to a class defined by axioms.
Example: `$\alpha :: \text{order}$`
- **schematic variables:** variables that can be instantiated.

Type Classes



→ similar to Haskell's type classes, but with semantic properties

```
class order =
```

```
  assumes order_refl: "x ≤ x"
```

```
  assumes order_trans: "⟦x ≤ y; y ≤ z⟧ ⇒ x ≤ z"
```

```
  ...
```

Type Classes



- similar to Haskell's type classes, but with semantic properties

```
class order =
```

```
  assumes order_refl: "x ≤ x"
```

```
  assumes order_trans: "⟦x ≤ y; y ≤ z⟧ ⇒ x ≤ z"
```

```
  ...
```

- theorems can be proved in the abstract

```
lemma order_less_trans: "∧ x :: 'a :: order. ⟦x < y; y < z⟧ ⇒ x < z"
```

Type Classes



- similar to Haskell's type classes, but with semantic properties

```
class order =
```

```
  assumes order_refl: "x ≤ x"
```

```
  assumes order_trans: "⟦x ≤ y; y ≤ z⟧ ⇒ x ≤ z"
```

```
  ...
```

- theorems can be proved in the abstract

```
lemma order_less_trans: "∧ x :: 'a :: order. ⟦x < y; y < z⟧ ⇒ x < z"
```

- can be used for subtyping

```
class linorder = order +
```

```
  assumes linorder_linear: "x ≤ y ∨ y ≤ x"
```

Type Classes



- similar to Haskell's type classes, but with semantic properties

```
class order =
```

```
  assumes order_refl: "x ≤ x"
```

```
  assumes order_trans: "⟦x ≤ y; y ≤ z⟧ ⇒ x ≤ z"
```

```
  ...
```

- theorems can be proved in the abstract

```
lemma order_less_trans: "∧ x :: 'a :: order. ⟦x < y; y < z⟧ ⇒ x < z"
```

- can be used for subtyping

```
class linorder = order +
```

```
  assumes linorder_linear: "x ≤ y ∨ y ≤ x"
```

- can be instantiated

```
instance nat :: "{order, linorder}" by ...
```

Schematic Variables



$$\frac{X \quad Y}{X \wedge Y}$$

→ X and Y must be **instantiated** to apply the rule

Schematic Variables



$$\frac{X \quad Y}{X \wedge Y}$$

→ X and Y must be **instantiated** to apply the rule

But: **lemma** “ $x + 0 = 0 + x$ ”

- x is free
- convention: lemma must be true for all x
- **during the proof**, x must **not** be instantiated

Schematic Variables



$$\frac{X \quad Y}{X \wedge Y}$$

→ X and Y must be **instantiated** to apply the rule

But: **lemma** “ $x + 0 = 0 + x$ ”

- x is free
- convention: lemma must be true for all x
- **during the proof**, x must **not** be instantiated

Solution:

Isabelle has **free** (x), **bound** (x), and **schematic** ($?X$) variables.

Only schematic variables can be instantiated.

Free converted into schematic after proof is finished.

Higher Order Unification



Unification:

Find substitution σ on variables for terms s, t such that $\sigma(s) = \sigma(t)$

Higher Order Unification



Unification:

Find substitution σ on variables for terms s, t such that $\sigma(s) = \sigma(t)$

In Isabelle:

Find substitution σ on schematic variables such that $\sigma(s) =_{\alpha\beta\eta} \sigma(t)$

Higher Order Unification



Unification:

Find substitution σ on variables for terms s, t such that $\sigma(s) = \sigma(t)$

In Isabelle:

Find substitution σ on schematic variables such that $\sigma(s) =_{\alpha\beta\eta} \sigma(t)$

Examples:

$$?X \wedge ?Y \quad =_{\alpha\beta\eta} \quad x \wedge x$$

$$?P \ x \quad =_{\alpha\beta\eta} \quad x \wedge x$$

$$P \ (?f \ x) \quad =_{\alpha\beta\eta} \quad ?Y \ x$$

Higher Order Unification



Unification:

Find substitution σ on variables for terms s, t such that $\sigma(s) = \sigma(t)$

In Isabelle:

Find substitution σ on schematic variables such that $\sigma(s) =_{\alpha\beta\eta} \sigma(t)$

Examples:

$$\begin{array}{lll} ?X \wedge ?Y & =_{\alpha\beta\eta} & x \wedge x \quad [?X \leftarrow x, ?Y \leftarrow x] \\ ?P \ x & =_{\alpha\beta\eta} & x \wedge x \quad [?P \leftarrow \lambda x. x \wedge x] \\ P \ (?f \ x) & =_{\alpha\beta\eta} & ?Y \ x \quad [?f \leftarrow \lambda x. x, ?Y \leftarrow P] \end{array}$$

Higher Order: schematic variables can be functions.

Higher Order Unification



→ Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable

Higher Order Unification



- Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- Unification modulo $\alpha\beta\eta$ is undecidable

Higher Order Unification



- Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- Unification modulo $\alpha\beta\eta$ is undecidable
- Higher Order Unification has possibly infinitely many solutions

Higher Order Unification



- Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- Unification modulo $\alpha\beta\eta$ is undecidable
- Higher Order Unification has possibly infinitely many solutions

But:

- Most cases are well-behaved

Higher Order Unification



- Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- Unification modulo $\alpha\beta\eta$ is undecidable
- Higher Order Unification has possibly infinitely many solutions

But:

- Most cases are well-behaved
- Important fragments (like Higher Order Patterns) are decidable

Higher Order Unification



- Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- Unification modulo $\alpha\beta\eta$ is undecidable
- Higher Order Unification has possibly infinitely many solutions

But:

- Most cases are well-behaved
- Important fragments (like Higher Order Patterns) are decidable

Higher Order Pattern:

- is a term in β normal form where
- each occurrence of a schematic variable is of the form $?f t_1 \dots t_n$
- and the $t_1 \dots t_n$ are η -convertible into n distinct bound variables

We have learned so far...



→ Simply typed lambda calculus: $\lambda \rightarrow$

We have learned so far...



- Simply typed lambda calculus: λ^{\rightarrow}
- Typing rules for λ^{\rightarrow} , type variables, type contexts

We have learned so far...



- Simply typed lambda calculus: λ^{\rightarrow}
- Typing rules for λ^{\rightarrow} , type variables, type contexts
- β -reduction in λ^{\rightarrow} satisfies subject reduction

We have learned so far...



- Simply typed lambda calculus: λ^{\rightarrow}
- Typing rules for λ^{\rightarrow} , type variables, type contexts
- β -reduction in λ^{\rightarrow} satisfies subject reduction
- β -reduction in λ^{\rightarrow} always terminates

We have learned so far...



- Simply typed lambda calculus: λ^{\rightarrow}
- Typing rules for λ^{\rightarrow} , type variables, type contexts
- β -reduction in λ^{\rightarrow} satisfies subject reduction
- β -reduction in λ^{\rightarrow} always terminates
- Types and terms in Isabelle