COMP4161
Advanced Topics in Software Verification

$$\lambda^{\rightarrow}$$

Gerwin Klein, Miki Tanaka, Johannes Åman Pohjola, Robert Sison

T3/2023

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 [4,5]
  - Datatype induction, primitive recursion [5,7]
  - General recursive functions, termination proofs [7[b]]
  - Proof automation, Isar (part 2) [8]
  - Hoare logic, proofs about programs, invariants [8,9]
  - C verification [9,10]
  - Practice, questions, exam prep [10[c]]

[a] a1 due; [b] a2 due; [c] a3 due

λ calculus is inconsistent

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

There are more terms that do not make sense:

$1\ 2$, $\texttt{true}\ \texttt{false}$, etc.

λ calculus is inconsistent

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

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* :: α

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 $(s\ t) :: \beta$

That’s about it

Now formally again

Syntax for $\lambda^{\rightarrow}$

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

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

Types: $\tau \ ::= \ \mathtt{b} \ | \ \nu \ | \ \tau \Rightarrow \tau$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

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

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

Syntax for $\lambda^{\rightarrow}$

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

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

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

Context Γ:
Γ: function from variable and constant names to types.

Syntax for $\lambda^{\rightarrow}$

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

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

Types: $\tau \quad ::= \quad \mathtt{b} \mid \nu \mid \tau \Rightarrow \tau$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

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

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

Context Γ:

Γ: function from variable and constant names to types.

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

Examples

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

Examples

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

$[y \leftarrow \texttt{int}] \vdash y ::$

Examples

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

$[y \leftarrow \texttt{int}] \vdash y :: \texttt{int}$

$[z \leftarrow \texttt{bool}] \vdash (\lambda y.\ y)\ z ::$

Examples

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

$[y \leftarrow \texttt{int}] \vdash y :: \texttt{int}$

$[z \leftarrow \texttt{bool}] \vdash (\lambda y.\ y)\ z :: \texttt{bool}$

$[] \vdash \lambda f\ x.\ f\ x ::$

Examples

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

$[y \leftarrow \texttt{int}] \vdash y :: \texttt{int}$

$[z \leftarrow \texttt{bool}] \vdash (\lambda y.\ y)\ z :: \texttt{bool}$

$[] \vdash \lambda f\ x.\ f\ x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$

Examples

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

$[y \leftarrow \texttt{int}] \vdash y :: \texttt{int}$

$[z \leftarrow \texttt{bool}] \vdash (\lambda y.\ y)\ z :: \texttt{bool}$

$[] \vdash \lambda f\ x.\ f\ x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$

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

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

Type Checking Rules

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

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

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

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

Type Checking Rules

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

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

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

Example Type Derivation:

$$\frac{}{[] \vdash \lambda x\ y.\ x ::}\ Abs$$

Remember:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

Example Type Derivation:

$$\dfrac{\dfrac{}{[x \leftarrow \alpha] \vdash \lambda y.\ x ::}\ Abs}{[] \vdash \lambda x\ y.\ x :: \alpha \Rightarrow}\ Abs$$

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

Example Type Derivation:

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

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

Example Type Derivation:

$$\cfrac{\cfrac{\cfrac{}{[x \leftarrow \alpha, y \leftarrow \beta] \vdash x :: \alpha}\ Var}{[x \leftarrow \alpha] \vdash \lambda y.\ x :: \beta \Rightarrow \alpha}\ Abs}{[] \vdash \lambda x\ y.\ x :: \alpha \Rightarrow \beta \Rightarrow \alpha}\ Abs$$

Remember:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

$$\frac{}{[] \vdash \lambda f\ x.\ f\ x\ x ::}\ Abs$$

Remember:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

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

Remember:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

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

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

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

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

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

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

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

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f\ x :: \alpha \Rightarrow \beta}\ App \quad \dfrac{}{\Gamma \vdash x :: \alpha}\ Var}{\Gamma \vdash f\ x\ x :: \beta}\ App}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x.\ f\ x\ x :: \alpha \Rightarrow \beta}\ Abs}{[] \vdash \lambda f\ x.\ f\ x\ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}\ Abs$$

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

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\ Var \quad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \quad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)}\ Var}{\Gamma \vdash f\ x :: \alpha \Rightarrow \beta}\ App \quad \overline{\Gamma \vdash x :: \alpha}\ Var}{\Gamma \vdash f\ x\ x :: \beta}\ App}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x.\ f\ x\ x :: \alpha \Rightarrow \beta}\ Abs}{[] \vdash \lambda f\ x.\ f\ x\ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}\ Abs$$

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

Remember:

$$\overline{\Gamma \vdash x :: \Gamma(x)}\ Var \qquad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ App \qquad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ Abs$$

More complex Example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)}\ \mathit{Var} \quad \dfrac{}{\Gamma \vdash x :: \alpha}\ \mathit{Var}}{\Gamma \vdash f\ x :: \alpha \Rightarrow \beta}\ \mathit{App} \quad \dfrac{}{\Gamma \vdash x :: \alpha}\ \mathit{Var}}{\dfrac{\Gamma \vdash f\ x\ x :: \beta}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x.\ f\ x\ x :: \alpha \Rightarrow \beta}\ \mathit{Abs}}\ \mathit{App}}{[] \vdash \lambda f\ x.\ f\ x\ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}\ \mathit{Abs}$$

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

Remember:

$$\dfrac{}{\Gamma \vdash x :: \Gamma(x)}\ \mathit{Var} \quad \dfrac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau}\ \mathit{App} \quad \dfrac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x.\ t) :: \tau_x \Rightarrow \tau}\ \mathit{Abs}$$

More general Types

A term can have more than one type.

More general Types

A term can have more than one type.

Example: $[] \vdash \lambda x.\ x :: \texttt{bool} \Rightarrow \texttt{bool}$
$[] \vdash \lambda x.\ x :: \alpha \Rightarrow \alpha$

More general Types

A term can have more than one type.

Example: $[] \vdash \lambda x.\ x :: \texttt{bool} \Rightarrow \texttt{bool}$
$[] \vdash \lambda x.\ x :: \alpha \Rightarrow \alpha$

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: $[] \vdash \lambda x.\ x :: \texttt{bool} \Rightarrow \texttt{bool}$
$[] \vdash \lambda x.\ x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

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

Examples:

$\texttt{int} \Rightarrow \texttt{bool} \quad \lesssim \quad \alpha \Rightarrow \beta$

More general Types

A term can have more than one type.

Example: $[] \vdash \lambda x.\ x :: \texttt{bool} \Rightarrow \texttt{bool}$
$[] \vdash \lambda x.\ x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

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

Examples:

$\texttt{int} \Rightarrow \texttt{bool} \quad \lesssim \quad \alpha \Rightarrow \beta \quad \lesssim \quad \beta \Rightarrow \alpha$

More general Types

A term can have more than one type.

Example: $[] \vdash \lambda x.\ x :: \texttt{bool} \Rightarrow \texttt{bool}$
$[] \vdash \lambda x.\ x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

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

Examples:

$$\texttt{int} \Rightarrow \texttt{bool} \quad \lesssim \quad \alpha \Rightarrow \beta \quad \lesssim \quad \beta \Rightarrow \alpha \quad \not\lesssim \quad \alpha \Rightarrow \alpha$$

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:

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

Most general Types

Fact: each type correct term has a most general type

Formally:

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

It can be found by executing the typing rules backwards.

Most general Types

Fact: each type correct term has a most general type

Formally:

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

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:

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

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:

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

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 $\lambda^{\rightarrow}$ 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: $\Gamma \vdash s :: \tau \ \wedge \ s \longrightarrow_\beta t \Longrightarrow \Gamma \vdash t :: \tau$

What about β reduction?

Definition of β reduction stays the same.

Fact: Well typed terms stay well typed during β reduction

Formally: $\Gamma \vdash s :: \tau \ \wedge \ s \longrightarrow_\beta t \Longrightarrow \Gamma \vdash t :: \tau$

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 $\longrightarrow_\beta$ 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 $\longrightarrow_\beta$ 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?

Checkpoint:

- ➜ untyped lambda calculus is turing complete
  (all computable functions can be expressed)
- ➜ but it is inconsistent
- ➜ $\lambda^{\rightarrow}$ "fixes" the inconsistency problem by adding types
- ➜ Problem: it is not turing complete anymore!

What does this mean for Expressiveness?

Checkpoint:

➜ untyped lambda calculus is turing complete
(all computable functions can be expressed)

➜ but it is inconsistent

➜ $\lambda^{\rightarrow}$ "fixes" the inconsistency problem by adding types

➜ Problem: it is not turing complete anymore!

Not all computable functions can be expressed in $\lambda^{\rightarrow}$!
(non terminating functions cannot be expressed)

What does this mean for Expressiveness?

Checkpoint:

- ➜ untyped lambda calculus is turing complete
  (all computable functions can be expressed)
- ➜ but it is inconsistent
- ➜ $\lambda^{\rightarrow}$ "fixes" the inconsistency problem by adding types
- ➜ Problem: it is not turing complete anymore!

Not all computable functions can be expressed in $\lambda^{\rightarrow}$!
(non terminating functions cannot be expressed)

But wait...

What does this mean for Expressiveness?

Checkpoint:

- ➜ untyped lambda calculus is turing complete
  (all computable functions can be expressed)
- ➜ but it is inconsistent
- ➜ $\lambda^{\rightarrow}$ "fixes" the inconsistency problem by adding types
- ➜ Problem: it is not turing complete anymore!

Not all computable functions can be expressed in $\lambda^{\rightarrow}$!
(non terminating functions cannot be expressed)

But wait... typed functional languages are turing complete!

What does this mean for Expressiveness?

So...

- ➜ typed functional languages are turing complete
- ➜ but $\lambda^{\rightarrow}$ is not...
- ➜ How does this work?

What does this mean for Expressiveness?

So...

- ➜ typed functional languages are turing complete
- ➜ but $\lambda^{\rightarrow}$ is not...
- ➜ How does this work?
- ➜ By adding one single constant, the Y operator (fix point operator), to $\lambda^{\rightarrow}$
- ➜ This introduces the non-termination that the types removed.

$$Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$$
$$Y\ t \longrightarrow_{\beta} t\ (Y\ t)$$

What does this mean for Expressiveness?

So...

- ➜ typed functional languages are turing complete
- ➜ but $\lambda^{\rightarrow}$ is not...
- ➜ How does this work?
- ➜ By adding one single constant, the Y operator (fix point operator), to $\lambda^{\rightarrow}$
- ➜ This introduces the non-termination that the types removed.

$$Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$$
$$Y\ t \longrightarrow_\beta t\ (Y\ t)$$

Fact: If we add Y to $\lambda^{\rightarrow}$ as the only constant, then each computable function can be encoded as closed, type correct $\lambda^{\rightarrow}$ term.

- ➜ Y is used for recursion
- ➜ lose decidability (what does $Y\ (\lambda x.\ x)$ reduce to?)
- ➜ (Isabelle/HOL doesn't have Y; recursion is more restricted)

Types and Terms in Isabelle

Types: $\tau \;::=\; \texttt{b} \mid {}'\nu \mid {}'\nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau)\ K$

$\texttt{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ base types

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

$K \in \{\texttt{set}, \texttt{list}, \ldots\}$ type constructors

$C \in \{\texttt{order}, \texttt{linord}, \ldots\}$ 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: $\tau \ ::= \ \mathtt{b} \ | \ '\nu \ | \ '\nu :: C \ | \ \tau \Rightarrow \tau \ | \ (\tau, \ldots, \tau)\ K$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

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

$K \in \{\mathtt{set}, \mathtt{list}, \ldots\}$ type constructors

$C \in \{\mathtt{order}, \mathtt{linord}, \ldots\}$ type classes

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

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

➜ **type constructors**: construct a new type out of a parameter type.
Example: `int list`

Types and Terms in Isabelle

Types: $\tau \;::=\; \mathtt{b} \mid {'\nu} \mid {'\nu :: C} \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau)\; K$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

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

$K \in \{\mathtt{set}, \mathtt{list}, \ldots\}$ type constructors

$C \in \{\mathtt{order}, \mathtt{linord}, \ldots\}$ 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: α :: *order*

Types and Terms in Isabelle

Types: $\tau \;::=\; \mathtt{b} \mid {'\nu} \mid {'\nu} :: C \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau)\; K$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

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

$K \in \{\mathtt{set}, \mathtt{list}, \ldots\}$ type constructors

$C \in \{\mathtt{order}, \mathtt{linord}, \ldots\}$ 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: $\alpha :: \mathit{order}$
- **schematic variables**: variables that can be instantiated.

Type Classes

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

  class order =

  assumes order_refl: "$x \leq x$"

  assumes order_trans: "$[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$"

  ...

Type Classes

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

  class order =

  assumes order_refl: "$x \leq x$"

  assumes order_trans: "$[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$"

  ...

- theorems can be proved in the abstract

  lemma order_less_trans: "$\bigwedge x ::'a :: \mathit{order}.\ [\![x < y; y < z]\!] \Longrightarrow x < z$"

Type Classes

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

  class order =

  assumes order_refl: "$x \leq x$"

  assumes order_trans: "$[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$"

  ...

- ➜ theorems can be proved in the abstract

  lemma order_less_trans: "$\bigwedge x :: 'a :: \mathit{order}.\ [\![x < y; y < z]\!] \Longrightarrow x < z$"

- ➜ can be used for subtyping

  class linorder = order +

  assumes linorder_linear: "$x \leq y \vee y \leq x$"

Type Classes

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

  class order =
      assumes order_refl: "$x \leq x$"
      assumes order_trans: "$[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$"
      ...

- ➜ theorems can be proved in the abstract

  lemma order_less_trans: "$\bigwedge x ::'a :: order.\ [\![x < y; y < z]\!] \Longrightarrow x < z$"

- ➜ can be used for subtyping

  class linorder = order +
      assumes linorder_linear: "$x \leq y \vee y \leq x$"

- ➜ 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:

$$
\begin{array}{lll}
?X \wedge ?Y & =_{\alpha\beta\eta} & x \wedge x \\
?P\ x & =_{\alpha\beta\eta} & x \wedge x \\
P\ (?f\ x) & =_{\alpha\beta\eta} & ?Y\ x
\end{array}
$$

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:

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

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 $?f\ t_1\ \ldots\ t_n$
- and the $t_1\ \ldots\ t_n$ 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: $\lambda^{\rightarrow}$
- ➜ Typing rules for $\lambda^{\rightarrow}$, type variables, type contexts

We have learned so far...

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

We have learned so far...

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

We have learned so far...

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