NICTA Advanced Course

Theorem Proving

Principles, Techniques, Applications

$$\lambda^{\rightarrow}$$

CONTENT

- ➜ Intro & motivation, getting started with Isabelle
- ➜ **Foundations & Principles**
  - **Lambda Calculus**
  - **Higher Order Logic, natural deduction**
  - Term rewriting
- ➜ Proof & Specification Techniques
  - Datatypes, recursion, induction
  - Inductively defined sets, rule induction
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

From last lecture:

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.

From last lecture:

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 $t :: \alpha$

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: $(\lambda x.\ x) :: \alpha \Rightarrow a$

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: $(\lambda x.\ x) :: \alpha \Rightarrow a$

➜ for $s\ t$ to be sensible:
s must be 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: $t \quad ::= \quad v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

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

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

$\mathsf{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ 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: $\tau \quad ::= \quad \mathsf{b} \mid \nu \mid \tau \Rightarrow \tau$

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

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

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

Contexts Γ:

Γ: function from variable and constant names to types.

Syntax for $\lambda^{\to}$

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

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

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

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

Contexts Γ:

Γ: function from variable and constant names to types.

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

Examples

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

Examples

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

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

Examples

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

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

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

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

Application:
$$\frac{}{\Gamma \vdash (t_1\ t_2) :: \tau_1}$$

Type Checking Rules

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

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

TYPE CHECKING RULES

Variables:

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

Application:

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

Abstraction:

$$\frac{}{\Gamma \vdash (\lambda x.\ t) :: \tau_1 \Rightarrow \tau_2}$$

TYPE CHECKING RULES

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

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

Abstraction: $$\frac{\Gamma[x \leftarrow \tau_1] \vdash t :: \tau_2}{\Gamma \vdash (\lambda x.\ t) :: \tau_1 \Rightarrow \tau_2}$$

EXAMPLE TYPE DERIVATION:

$$[\,] \vdash \lambda x\ y.\ x :: \alpha \Rightarrow \beta \Rightarrow \alpha$$

Example Type Derivation:

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

Example Type Derivation:

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

EXAMPLE TYPE DERIVATION:

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

More complex Example

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

MORE COMPLEX EXAMPLE

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

More complex Example

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

More complex Example

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

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

More complex Example

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

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

More complex Example

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

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

More complex Example

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

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

More complex Example

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

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

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

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?

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

What does this mean for Expressiveness?

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

How can typed functional languages then be turing complete?

WHAT DOES THIS MEAN FOR EXPRESSIVENESS?

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

How can typed functional languages then be turing complete?

Fact:
Each computable function can be encoded as closed, type correct $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$!

How can typed functional languages then be turing complete?

Fact:
Each computable function can be encoded as closed, type correct $\lambda^{\rightarrow}$ 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

Types and Terms in Isabelle

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

$\mathsf{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ 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: $t \quad ::= \quad v \mid c \mid ?v \mid (t\ t) \mid (\lambda x.\ t)$

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

Types and Terms in Isabelle

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

$\mathsf{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ 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: $t \quad ::= \quad v \mid c \mid ?v \mid (t\ t) \mid (\lambda x.\ t)$

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

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

Types and Terms in Isabelle

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

$\mathsf{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ 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

➜ **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: $\alpha :: order$

Types and Terms in Isabelle

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

$\mathsf{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ 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

➜ **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: $\alpha :: order$

➜ **schematic variables**: variables that can be instantiated.

TYPE CLASSES

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

axclass order $<$ ord

order_refl: "$x \leq x$"

order_trans: "$\llbracket x \leq y; y \leq z \rrbracket \Longrightarrow x \leq z$"

$\ldots$

TYPE CLASSES

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

  axclass order $<$ ord

  order_refl: "$x \le x$"

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

  ...

- theorems can be proved in the abstract

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

TYPE CLASSES

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

axclass order $<$ ord

order_refl: "$x \leq x$"

order_trans: "$\llbracket x \leq y; y \leq z \rrbracket \Longrightarrow x \leq z$"

$\ldots$

➜ theorems can be proved in the abstract

lemma order_less_trans: "$\bigwedge x ::'a :: order.\ \llbracket x < y; y < z \rrbracket \Longrightarrow x < z$"

➜ can be used for subtyping

axclass linorder $<$ order

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

TYPE CLASSES

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

axclass order $<$ ord

order_refl: "$x \leq x$"

order_trans: "$\llbracket x \leq y; y \leq z \rrbracket \Longrightarrow x \leq z$"

...

➜ theorems can be proved in the abstract

lemma order_less_trans: "$\bigwedge x :: 'a :: order.\ \llbracket x < y; y < z \rrbracket \Longrightarrow x < z$"

➜ can be used for subtyping

axclass linorder $<$ order

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 fi nished.

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:

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

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

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

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 from $?f\ t_1\ \dots\ t_n$
- 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: $\lambda^{\rightarrow}$
- 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

Preview: Proofs in Isabelle

General schema:

lemma name: ”<goal>”

apply <method>

apply <method>

. . .

done

General schema:

lemma name: ”<goal>”

apply <method>

apply <method>

. . .

done

➜ Sequential application of methods until all **subgoals** are solved.

The Proof State

1. $\bigwedge x_1 \ldots x_p. [\![A_1; \ldots ; A_n]\!] \Longrightarrow B$

2. $\bigwedge y_1 \ldots y_q. [\![C_1; \ldots ; C_m]\!] \Longrightarrow D$

THE PROOF STATE

1. $\bigwedge x_1 \dots x_p . [\![A_1; \dots ; A_n]\!] \Longrightarrow B$

2. $\bigwedge y_1 \dots y_q . [\![C_1; \dots ; C_m]\!] \Longrightarrow D$

$x_1 \dots x_p$	Parameters
$A_1 \dots A_n$	Local assumptions
B	Actual (sub)goal

ISABELLE THEORIES

Syntax:

`theory` $MyTh$ = $ImpTh_1$ `+ ... +` $ImpTh_n$ `:`

(declarations, defi nitions, theorems, proofs, ...)*

`end`

➜ $MyTh$: name of theory. Must live in file $MyTh$`.thy`

➜ $ImpTh_i$: name of *imported* theories. Import transitive.

ISABELLE THEORIES

Syntax:

`theory` $MyTh$ = $ImpTh_1$ + ... + $ImpTh_n$:

(declarations, definitions, theorems, proofs, ...)*

`end`

➜ $MyTh$: name of theory. Must live in file $MyTh$`.thy`

➜ $ImpTh_i$: name of *imported* theories. Import transitive.

Unless you need something special:

`theory` $MyTh$ `= Main:`

NATURAL DEDUCTION RULES

$$\frac{}{A \wedge B}\ \text{conjI} \qquad \frac{A \wedge B}{C}\ \text{conjE}$$

$$\frac{}{A \vee B}\ \frac{}{A \vee B}\ \text{disjI1/2} \qquad \frac{A \vee B}{C}\ \text{disjE}$$

$$\frac{}{A \longrightarrow B}\ \text{impI} \qquad \frac{A \longrightarrow B}{C}\ \text{impE}$$

For each connective ($\wedge, \vee$, etc):

introduction and **elemination** rules

NATURAL DEDUCTION RULES

$$\frac{A \quad B}{A \wedge B}\ \text{conjI} \qquad \frac{A \wedge B}{C}\ \text{conjE}$$

$$\frac{}{A \vee B} \quad \frac{}{A \vee B}\ \text{disjI1/2} \qquad \frac{A \vee B}{C}\ \text{disjE}$$

$$\frac{}{A \longrightarrow B}\ \text{impI} \qquad \frac{A \longrightarrow B}{C}\ \text{impE}$$

For each connective ($\wedge, \vee$, etc):

introduction and **elemination** rules

NATURAL DEDUCTION RULES

$$\frac{A \quad B}{A \wedge B} \text{ conjI} \qquad \frac{A \wedge B \quad [\![A; B]\!] \Longrightarrow C}{C} \text{ conjE}$$

$$\frac{}{A \vee B} \quad \frac{}{A \vee B} \text{ disjI1/2} \qquad \frac{A \vee B}{C} \text{ disjE}$$

$$\frac{}{A \longrightarrow B} \text{ impI} \qquad \frac{A \longrightarrow B}{C} \text{ impE}$$

For each connective ($\wedge, \vee$, etc):
introduction and **elemination** rules

NATURAL DEDUCTION RULES

$$\frac{A \quad B}{A \wedge B}\ \text{conjI} \qquad \frac{A \wedge B \quad [\![A; B]\!] \Longrightarrow C}{C}\ \text{conjE}$$

$$\frac{A}{A \vee B} \quad \frac{B}{A \vee B}\ \text{disjI1/2} \qquad \frac{A \vee B}{C}\ \text{disjE}$$

$$\frac{}{A \longrightarrow B}\ \text{impI} \qquad \frac{A \longrightarrow B}{C}\ \text{impE}$$

For each connective ($\wedge, \vee$, etc):
introduction and **elemination** rules

NATURAL DEDUCTION RULES

$$\frac{A \quad B}{A \wedge B} \text{ conjI} \qquad \frac{A \wedge B \quad [\![A; B]\!] \Longrightarrow C}{C} \text{ conjE}$$

$$\frac{A}{A \vee B} \; \frac{B}{A \vee B} \text{ disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C} \text{ disjE}$$

$$\frac{}{A \longrightarrow B} \text{ impI} \qquad \frac{A \longrightarrow B}{C} \text{ impE}$$

For each connective ($\wedge, \vee$, etc):

introduction and **elemination** rules

NATURAL DEDUCTION RULES

$$\frac{A \quad B}{A \wedge B}\ \text{conjI} \qquad \frac{A \wedge B \quad [\![A; B]\!] \Longrightarrow C}{C}\ \text{conjE}$$

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \text{disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \text{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \text{impI} \qquad \frac{A \longrightarrow B}{C}\ \text{impE}$$

For each connective ($\wedge, \vee$, etc):

introduction and **elemination** rules

NATURAL DEDUCTION RULES

$$\frac{A \quad B}{A \wedge B}\ \text{conjI} \qquad \frac{A \wedge B \quad [\![A; B]\!] \Longrightarrow C}{C}\ \text{conjE}$$

$$\frac{A}{A \vee B}\ \frac{B}{A \vee B}\ \text{disjI1/2} \qquad \frac{A \vee B \quad A \Longrightarrow C \quad B \Longrightarrow C}{C}\ \text{disjE}$$

$$\frac{A \Longrightarrow B}{A \longrightarrow B}\ \text{impI} \qquad \frac{A \longrightarrow B \quad A \quad B \Longrightarrow C}{C}\ \text{impE}$$

For each connective ($\wedge, \vee$, etc):

introduction and **elemination** rules

Proof by assumption

apply assumption

proves

1. $[\![B_1; \ldots; B_m]\!] \Longrightarrow C$

by unifying C with one of the B_i

apply assumption

proves

1. $[\![B_1; \ldots; B_m]\!] \Longrightarrow C$

by unifying C with one of the B_i

There may be more than one matching B_i and multiple unifi ers.

Backtracking!

Explicit backtracking command: **back**

INTRO RULES

Intro rules decompose formulae to the right of $\Longrightarrow$.

apply (rule $<$intro-rule$>$)

INTRO RULES

Intro rules decompose formulae to the right of $\Longrightarrow$.

apply (rule <intro-rule>)

Intro rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ means

- ➜ To prove A it suffices to show $A_1 \ldots A_n$

INTRO RULES

Intro rules decompose formulae to the right of $\Longrightarrow$.

apply (rule <intro-rule>)

Intro rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ means

- ➜ To prove A it suffices to show $A_1 \ldots A_n$

Applying rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ to subgoal C:

- ➜ unify A and C
- ➜ replace C with n new subgoals $A_1 \ldots A_n$

ELIM RULES

Elim rules decompose formulae on the left of $\Longrightarrow$.

apply (erule $<$elim-rule$>$)

ELIM RULES

Elim rules decompose formulae on the left of $\Longrightarrow$.

apply (erule <elim-rule>)

Elim rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ means

➜ If I know A_1 and want to prove A it suffices to show $A_2 \ldots A_n$

ELIM RULES

Elim rules decompose formulae on the left of $\Longrightarrow$.

apply (erule $<$elim-rule$>$)

Elim rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ means

➜ If I know A_1 and want to prove A it suffices to show $A_2 \ldots A_n$

Applying rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ to subgoal C:
Like **rule** but also

➜ unifies first premise of rule with an assumption

➜ eliminates that assumption

DEMO

EXERCISES

- ➜ what are the types of $\lambda x\ y.\ y\ x$ and $\lambda x\ y\ z.\ x\ y\ (y\ z)$
- ➜ construct a type derivation tree on paper for $\lambda x\ y\ z.\ x\ y\ (y\ z)$
- ➜ find a unifier (substitution) such that $\lambda x\ y.\ ?F\ x = \lambda x\ y.\ c\ (?G\ y\ x)$
- ➜ prove $(A \longrightarrow B \longrightarrow C) = (A \wedge B \longrightarrow C)$ in Isabelle
- ➜ prove $\neg(A \wedge B) \Longrightarrow \neg A \vee \neg B$ in Isabelle (tricky!)