NICTA Advanced Course

Theorem Proving
Principles, Techniques, Applications
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
\textbf{\La} \textsc{CAlculus IS INCONSISTENT}

\begin{itemize}
  \item From last lecture:
    \begin{enumerate}
      \item Can find term $R$ such that $R \ R \ \Rightarrow_{\beta} \ \text{not}(R \ R)$
    \end{enumerate}
  \item There are more terms that do not make sense:
    \begin{itemize}
      \item 1 2, true false, etc.
    \end{itemize}
\end{itemize}
From last lecture:
Can find term $R$ such that $R \ R \equiv_{\beta} \ \text{not}(R \ R)$

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

**Examples:**
**Idea:** assign a type to each “sensible” $\lambda$ term.

**Examples:**
- for term $t$ has type $\alpha$ write $t :: \alpha$
**INTRODUCING TYPES**

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

**Examples:**

→ for *term t has type* \( \alpha \) write \( t :: \alpha \)

→ if \( x \) has type \( \alpha \) then \( \lambda x. x \) is a function from \( \alpha \) to \( \alpha \)

Write: \( (\lambda x. x) :: \alpha \Rightarrow a \)
**Idea:** assign a type to each “sensible” \( \lambda \) term.

**Examples:**

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

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

**Types:**
\[
\tau \ ::= \ b \mid \nu \mid \tau \to \tau
\]
\(b \in \{\text{bool, int, ...}\}\) base types
\(\nu \in \{\alpha, \beta, ...\}\) type variables

\[
\alpha \to \beta \to \gamma = \alpha \to (\beta \to \gamma)
\]
**Syntax for \( \lambda \rightarrow \)**

**Terms:**

\[
t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t)
\]

where \( v, x \in V, \ c \in C, \ V, C \) sets of names.

**Types:**

\[
\tau ::= b \mid v \mid \tau \Rightarrow \tau
\]

where \( b \in \{\text{bool, int, \ldots}\} \) base types

\( \nu \in \{\alpha, \beta, \ldots\} \) type variables

\[
\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)
\]

**Contexts** \( \Gamma \):

\( \Gamma \): function from variable and constant names to types.
**Syntax for** λ→

**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:** \( \tau ::= b \mid v \mid \tau \Rightarrow \tau \)
\( b \in \{\text{bool, int, ...}\} \) base types
\( v \in \{\alpha, \beta, ...\} \) type variables

\[
\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)
\]

**Contexts** \( \Gamma \):

\( \Gamma \): function from variable and constant names to types.

**Term** \( t \) **has type** \( \tau \) **in context** \( \Gamma \):

\( \Gamma \vdash t :: \tau \)
Γ ⊢ (λx. x) :: α ⇒ α
$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$

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

\[ [y \leftarrow \text{int}] \vdash y :: \text{int} \]

\[ [z \leftarrow \text{bool}] \vdash (\lambda y. y) z :: \text{bool} \]
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 \( \Gamma \) and \( \tau \) such that \( \Gamma \vdash t :: \tau \)
**Type Checking Rules**

**Variables:**

\[ \Gamma \vdash x :: \Gamma(x) \]
TYPE CHECKING RULES

Variables:

\[ \Gamma \vdash x :: \Gamma(x) \]

Application:

\[ \Gamma \vdash (t_1 \ t_2) :: \tau_1 \]
**TYPE CHECKING RULES**

Variables: \[ \Gamma \vdash x :: \Gamma(x) \]

Application: \[ \Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau_1 \quad \Gamma \vdash t_2 :: \tau_2 \]
\[ \Gamma \vdash (t_1 \ t_2) :: \tau_1 \]
TYPE CHECKING RULES

Variables: \[ \Gamma \vdash x :: \Gamma(x) \]

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: \[
\frac{\Gamma \vdash (\lambda x. t) :: \tau_1 \Rightarrow \tau_2}{\Gamma \vdash (\lambda x. t) :: \tau_1} \]
**Type Checking Rules**

**Variables:**

\[ \Gamma \vdash x :: \Gamma(x) \]

**Application:**

\[ \Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau_1 \quad \Gamma \vdash t_2 :: \tau_2 \]

\[ \Gamma \vdash (t_1 \ t_2) :: \tau_1 \]

**Abstraction:**

\[ \Gamma[x \leftarrow \tau_1] \vdash t :: \tau_2 \]

\[ \Gamma \vdash (\lambda x. \ t) :: \tau_1 \Rightarrow \tau_2 \]
EXAMPLE TYPE DERIVATION:


d \vdash \lambda x \ y. \ x :: \alpha \Rightarrow \beta \Rightarrow \alpha
EXAMPLE TYPE DERIVATION:

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

\[ [] \vdash \lambda x \ y. x :: \alpha \Rightarrow \beta \Rightarrow \alpha \]
EXAMPLE TYPE DERIVATION:

\[
\begin{align*}
\vdash x &\quad \text{[}x \leftarrow \alpha, y \leftarrow \beta\text{]} \vdash x :: \alpha \\
\vdash x &\quad \text{[}x \leftarrow \alpha\text{]} \vdash \lambda y. x :: \beta \Rightarrow \alpha \\
\vdash \lambda x y. x :: \alpha \Rightarrow \beta \Rightarrow \alpha
\end{align*}
\]
EXAMPLE TYPE DERIVATION:

\[
\begin{align*}
[x \leftarrow \alpha, y \leftarrow \beta] & \vdash x :: \alpha \\
[x \leftarrow \alpha] & \vdash \lambda y. x :: \beta \Rightarrow \alpha \\
[] & \vdash \lambda x \ y. x :: \alpha \Rightarrow \beta \Rightarrow \alpha
\end{align*}
\]
More complex example

\[ [] \vdash \lambda f \, x. f \, x \, x :: \]
\[ \left[ \right \vdash \lambda f \, x. \, f \, x \, x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \]
[\[ f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta \] \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
\[ \Gamma \vdash f \, x \, x :: \beta \]

\[ [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. \, f \, x \, x :: \alpha \Rightarrow \beta \]

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

\[ \Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, \, x \leftarrow \alpha] \]
\[
\Gamma \vdash f \ x :: \alpha \Rightarrow \beta \\
\quad
\Gamma \vdash f \ x \ x :: \beta \\
\quad
[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. \ f \ x \ x :: \alpha \Rightarrow \beta \\
\quad
\emptyset \vdash \lambda f. \ f \ x \ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta
\]

\[
\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, \ x \leftarrow \alpha]
\]
\[
\begin{align*}
\Gamma & \vdash f \cdot x :: \alpha \Rightarrow \beta & \Gamma & \vdash x :: \alpha \\
\Gamma & \vdash f \cdot x \cdot x :: \beta \\
[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] & \vdash \lambda x. f \cdot x \cdot x :: \alpha \Rightarrow \beta \\
[] & \vdash \lambda f \cdot x. f \cdot x \cdot x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta
\end{align*}
\]

\[
\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]
\]
MORE COMPLEX EXAMPLE

\[
\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta) \\
\Gamma \vdash f \; x :: \alpha \Rightarrow \beta \\
\Gamma \vdash x :: \alpha \\
\Gamma \vdash f \; x \; x :: \beta \\
[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. \; f \; x \; x :: \alpha \Rightarrow \beta \\
[] \vdash \lambda f \; x. \; f \; x \; x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta
\]

\[
\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]
\]
\[ \Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta) \quad \Gamma \vdash x :: \alpha \]

\[ \Gamma \vdash f \, x :: \alpha \Rightarrow \beta \quad \Gamma \vdash x :: \alpha \]

\[ \Gamma \vdash f \, x \, x :: \beta \quad \Gamma \vdash x :: \alpha \]

\[ [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. \ f \, x \, x :: \alpha \Rightarrow \beta \]

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

\[ \Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, \ x \leftarrow \alpha] \]
A term can have more than one type.
MORE GENERAL TYPES

A term can have more than one type.

Example: \[ \emptyset \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool} \]
\[ \emptyset \vdash \lambda x. x :: \alpha \Rightarrow \alpha \]
MORE GENERAL TYPES

A term can have more than one type.

Example:  [] ⊢ λx. x :: bool ⇒ bool

            [] ⊢ λx. x :: α ⇒ α

Some types are more general than others:

τ ≤ σ  if there is a substitution S such that  τ = S(σ)
**More General Types**

A term can have more than one type.

**Example:**

\[
[] \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}
\]

\[
[] \vdash \lambda x. x :: \alpha \Rightarrow \alpha
\]

Some types are more general than others:

\[\tau \preceq \sigma\] if there is a substitution \(S\) such that \(\tau = S(\sigma)\)

**Examples:**

\[\text{int} \Rightarrow \text{bool} \preceq \alpha \Rightarrow \beta\]
A term can have more than one type.

**Example:**  
\[ [] \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool} \]
\[ [] \vdash \lambda x. x :: \alpha \Rightarrow \alpha \]

Some types are more general than others:

\[ \tau \preceq \sigma \quad \text{if there is a substitution } S \text{ such that } \tau = S(\sigma) \]

**Examples:**

\[ \text{int} \Rightarrow \text{bool} \preceq \alpha \Rightarrow \beta \preceq \beta \Rightarrow \alpha \]
A term can have more than one type.

Example: \[ \emptyset \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool} \]
\[ \emptyset \vdash \lambda x. x :: \alpha \Rightarrow \alpha \]

Some types are more general than others:
\[ \tau \preceq \sigma \] if there is a substitution \( S \) such that \( \tau = S(\sigma) \)

Examples:
\[ \text{int} \Rightarrow \text{bool} \preceq \alpha \Rightarrow \beta \preceq \beta \Rightarrow \alpha \not\preceq \alpha \Rightarrow \alpha \]
Fact: each type correct term has a most general type
**MOST general TYPES**

**Fact:** each type correct term has a most general type

**Formally:**
\[
\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \land (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \preceq \sigma)
\]
**Most General Types**

**Fact:** each type correct term has a most general type

**Formally:**
\[ \Gamma \vdash t :: \tau \quad \rightarrow \quad \exists \sigma. \Gamma \vdash t :: \sigma \land (\forall \sigma'. \Gamma \vdash t :: \sigma' \Rightarrow \sigma' \preceq \sigma) \]

It can be found by executing the typing rules backwards.
**MOST GENERAL TYPES**

**Fact:** each type correct term has a most general type

**Formally:**

\[ \Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \land (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \lesssim \sigma) \]

It can be found by executing the typing rules backwards.

→ **type checking:** checking if \( \Gamma \vdash t :: \tau \) for given \( \Gamma \) and \( \tau \)
Fact: each type correct term has a most general type

Formally:
\[
\Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \land (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \preceq \sigma)
\]

It can be found by executing the typing rules backwards.

→ type checking: checking if \( \Gamma \vdash t :: \tau \) for given \( \Gamma \) and \( \tau \)

→ type inference: computing \( \Gamma \) and \( \tau \) such that \( \Gamma \vdash t :: \tau \)
**Fact:** each type correct term has a most general type

**Formally:**
\[ \Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \land (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \sqsubseteq \sigma) \]

It can be found by executing the typing rules backwards.

→ **type checking:** checking if \( \Gamma \vdash t :: \tau \) for given \( \Gamma \) and \( \tau \)

→ **type inference:** computing \( \Gamma \) and \( \tau \) such that \( \Gamma \vdash t :: \tau \)

Type checking and type inference on \( \lambda \rightarrow \) are decidable.
Definition of reduction stays the same.

Fact:

Well-typed terms stay well-typed during reduction.

Formally:

\[ \text{\texttt{s}} :: \downarrow \text{\texttt{s}}^! \text{\texttt{t}} = \text{\texttt{t}} :: \downarrow \text{\texttt{t}} \]

This property is called subject reduction.
WHAT ABOUT $\beta$ REDUCTION?

Definition of $\beta$ reduction stays the same.
Definition of $\beta$ reduction stays the same.

**Fact:** Well typed terms stay well typed during $\beta$ reduction

**Formally:** $\Gamma \vdash s :: \tau \land s \rightarrow^\beta t \implies \Gamma \vdash t :: \tau$
WHAT ABOUT $\beta$ REDUCTION?

Definition of $\beta$ reduction stays the same.

Fact: Well typed terms stay well typed during $\beta$ reduction

Formally: \[ \Gamma \vdash s :: \tau \land s \rightarrow_\beta t \implies \Gamma \vdash t :: \tau \]

This property is called subject reduction
What about termination?

Alan Turing, 1942

- Is decidable to decide if $s = t$, reduce $s$ and $t$ to normal form (always exists, because it terminates), and compare result.

This is why Isabelle can automatically reduce each term to normal form.
WHAT ABOUT TERMINATION?

\[ \beta \text{ reduction in } \lambda \rightarrow \text{ always terminates.} \]

(Alan Turing, 1942)
WHAT ABOUT TERMINATION?

\(\beta\) reduction in \(\lambda\rightarrow\) always terminates.

(Alan Turing, 1942)

\[\Rightarrow \; =_\beta \; \text{is decidable}\]

To decide if \(s =_\beta t\), reduce \(s\) and \(t\) to normal form (always exists, because \(\rightarrow\beta\) terminates), and compare result.
WHAT ABOUT TERMINATION?

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

(Alan Turing, 1942)

$\Rightarrow \; =_\beta$ is decidable
To decide if $s =_\beta t$, reduce $s$ and $t$ to normal form (always exists, because $\longrightarrow_\beta$ terminates), and compare result.

$\Rightarrow \; =_{\alpha\beta\eta}$ is decidable
This is why Isabelle can automatically reduce each term to $\beta\eta$ normal form.
WHAT DOES THIS MEAN FOR EXPRESSIVENESS?
Not all computable functions can be expressed in $\lambda \rightarrow$!
Not all computable functions can be expressed in $\lambda \rightarrow$!

How can typed functional languages then be turing complete?
Not all computable functions can be expressed in $\lambda \to$!

How can typed functional languages then be turing complete?

**Fact:**
Each computable function can be encoded as closed, type correct $\lambda \to$ term using $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$ with $Y \ t \to_\beta t \ (Y \ t)$ as only constant.
What does this mean for expressiveness?

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

How can typed functional languages then be turing complete?

**Fact:**
Each computable function can be encoded as closed, type correct $\lambda \to$ term using $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$ with $Y \, t \to_{\beta} t \,(Y \, t)$ as only constant.

$\Rightarrow Y$ is called fix point operator

$\Rightarrow$ used for recursion
Types and Terms in Isabelle

Types: \[ \tau ::= \mathit{b} \mid \nu \mid \nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau) \mathit{K} \]
- \( \mathit{b} \in \{\text{bool, int, \ldots}\} \) base types
- \( \nu \in \{\alpha, \beta, \ldots\} \) type variables
- \( \mathit{K} \in \{\text{set, list, \ldots}\} \) type constructors
- \( \mathit{C} \in \{\text{order, linord, \ldots}\} \) type classes

Terms: \[ t ::= v \mid c \mid ?v \mid (t \ t) \mid (\lambda x. \ t) \]
- \( v, x \in V, \ c \in C, \ V, C \) sets of names
Types and Terms in Isabelle

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

- $b \in \{\text{bool, int, \ldots}\}$ base types
- $v \in \{\alpha, \beta, \ldots\}$ type variables
- $K \in \{\text{set, list, \ldots}\}$ type constructors
- $C \in \{\text{order, linord, \ldots}\}$ type classes

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

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

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

Example: int list
Types: \( \tau ::= \ b \ | \ '\nu \ | \ '\nu :: C \ | \ \tau \Rightarrow \tau \ | \ (\tau, \ldots, \tau) \ K \)

- \( b \in \{\text{bool, int,} \ldots\} \) base types
- \( \nu \in \{\alpha, \beta, \ldots\} \) type variables
- \( K \in \{\text{set, list,} \ldots\} \) type constructors
- \( C \in \{\text{order, linord,} \ldots\} \) type classes

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

- \( v, x \in V, \ c \in C, \ V, C \) sets of names

- **type constructors**: construct a new type out of a parameter type. Example: \( \text{int list} \)

- **type classes**: restrict type variables to a class defined by axioms. Example: \( \alpha :: \text{order} \)
**Types and Terms in Isabelle**

**Types:**

\[ \tau ::= b \mid \nu \mid \nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau) \]

- \(b \in \{\text{bool, int,} \ldots\}\) base types
- \(\nu \in \{\alpha, \beta, \ldots\}\) type variables
- \(K \in \{\text{set, list,} \ldots\}\) type constructors
- \(C \in \{\text{order, linord,} \ldots\}\) type classes

**Terms:**

\[ t ::= v \mid c \mid \nu \mid (tt) \mid (\lambda x. t) \]

- \(v, x \in V, \ c \in C, \ V, C\) sets of names

→ **type constructors:** construct a new type out of a parameter type.
   *Example:* \(\text{int list}\)

→ **type classes:** restrict type variables to a class defined by axioms.
   *Example:* \(\alpha :: \text{order}\)

→ **schematic variables:** variables that can be instantiated.
similar to Haskell’s type classes, but with semantic properties

```haskell
axclass order < ord
  order_refl: "x ≤ x"
  order_trans: "[x ≤ y; y ≤ z] → x ≤ z"
  ...
```
similar to Haskell’s type classes, but with semantic properties

axclass order $<$ ord

order_refl: $”x \leq x”$

order_trans: $”[x \leq y; y \leq z] \implies x \leq z”$

...

theorems can be proved in the abstract

lemma order_less_trans: $”\forall x ::’a :: order. [x < y; y < z] \implies x < z”$
similar to Haskell’s type classes, but with semantic properties

**axclass** order $<$ ord
- order_refl: $”x \leq x”$
- order_trans: $”[x \leq y; y \leq z] \implies x \leq z”$
  
... 

theorems can be proved in the abstract

**lemma** order_less_trans: $”\forall x ::'a :: order. [x < y; y < z] \implies x < z”$

can be used for subtyping

**axclass** linorder $<$ order
- linorder_linear: $”x \leq y \lor y \leq x”$
→ similar to Haskell’s type classes, but with semantic properties
   
   **axclass** order $<$ ord
   
   order_refl: "$x \leq x$"
   
   order_trans: "[$x \leq y; y \leq z] \implies x \leq z"
   
   ... 

   → theorems can be proved in the abstract

   **lemma** order_less_trans: "$\forall x ::'a :: order. [x < y; y < z] \implies x < z$"

   → can be used for subtyping

   **axclass** linorder $<$ order

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

   → can be instantiated

   **instance** nat :: "{"order, linorder}" by ...
**Schematic Variables**

\[
\frac{X \quad Y}{X \land Y}
\]

→ \(X\) and \(Y\) must be **instantiated** to apply the rule
Schematic Variables

\[
\frac{X \quad Y}{X \land Y}
\]

→ \(X\) and \(Y\) must be \textit{instantiated} to apply the rule

But: \textbf{lemma } ”\(x + 0 = 0 + x”\)

→ \(x\) is free
→ convention: lemma must be true for all \(x\)
→ \textit{during the proof}, \(x\) must \textbf{not} be instantiated
**SCHEMATIC VARIABLES**

\[
\begin{array}{c|c}
X & Y \\
\hline
X \wedge Y \\
\end{array}
\]

→ $X$ 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.
Unification:
Find substitution $\sigma$ on variables for terms $s, t$ such that $\sigma(s) = \sigma(t)$
Higher Order Unification

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

In Isabelle:
Find substitution $\sigma$ on schematic variables such that $\sigma(s) =_{\alpha \beta \eta} \sigma(t)$
Higher Order Unification

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

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

Examples:

\[
\begin{align*}
?X \land ?Y &=_{\alpha\beta\eta} x \land x \\
?P \ x &=_{\alpha\beta\eta} x \land x \\
P \ (?f \ x) &=_{\alpha\beta\eta} ?Y \ x
\end{align*}
\]
**Higher Order Unification**

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

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

Examples:
\[
\begin{align*}
?X \land ?Y &=_{\alpha\beta\eta} x \land x & [?X \leftarrow x, ?Y \leftarrow x] \\
?P \ x &=_{\alpha\beta\eta} x \land x & [?P \leftarrow \lambda x. \ x \land x] \\
P \ (?f \ x) &=_{\alpha\beta\eta} ?Y \ x & [?f \leftarrow \lambda x. \ x, ?Y \leftarrow P]
\end{align*}
\]

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 Pattern:
- is a term normal form where
  - each occurrence of a schematic variable is of the form $\text{?}f_1:::t_n$
  - and the $t_1:::t_n$ are $\eta$-convertible into $n$ distinct bound variables
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 $\beta$ 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 $\eta$-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
- \( \beta \)-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
→ $\beta$-reduction in $\lambda \rightarrow$ satisfies subject reduction
→ $\beta$-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
- $\beta$-reduction in $\lambda \rightarrow$ satisfies subject reduction
- $\beta$-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:

```plaintext
lemma name: "<goal>"
apply <method>
apply <method>
...
done
```

→ Sequential application of methods until all subgoals are solved.
**THE PROOF STATE**

1. \( \bigwedge x_1 \cdots x_p. [A_1; \ldots; A_n] \implies B \)

2. \( \bigwedge y_1 \cdots y_q. [C_1; \ldots; C_m] \implies D \)
THE PROOF STATE

1. \( \land x_1 \ldots x_p. [A_1; \ldots; A_n] \Rightarrow B \)
2. \( \land y_1 \ldots y_q. [C_1; \ldots; C_m] \Rightarrow D \)

\( x_1 \ldots x_p \) Parameters
\( A_1 \ldots A_n \) Local assumptions
\( B \) Actual (sub)goal
ISABELLE THEORIES

Syntax:

\texttt{theory \textit{MyTh} = \textit{ImpTh}_1 + \ldots + \textit{ImpTh}_n:}

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

end

\rightarrow \textit{MyTh}: name of theory. Must live in file \texttt{MyTh.thy}

\rightarrow \textit{ImpTh}_i: name of imported theories. Import transitive.
Syntax:

\[
\text{theory } M\text{y}Th = I\text{mp}Th_1 + \ldots + I\text{mp}Th_n:
\]
(declarations, definitions, theorems, proofs, ...)*
end

→ \textit{MyTh}: name of theory. Must live in file \textit{MyTh.thy}

→ \textit{ImpTh}_i: name of imported theories. Import transitive.

Unless you need something special:

\[
\text{theory } M\text{y}Th = \text{Main}:
\]
**NATURAL DEDUCTION RULES**

\[
\begin{align*}
A \land B & \quad \text{conjI} \quad A \land B \quad \text{conjE} \\
A \lor B & \quad \text{disjI1/2} \quad A \lor B \quad \text{disjE} \\
A \rightarrow B & \quad \text{impl} \quad A \rightarrow B \quad \text{impE}
\end{align*}
\]

For each connective (\(\land, \lor, \text{etc}\)): introduction and elimination rules
**Natural Deduction Rules**

For each connective (\(\land, \lor, \text{etc}\):)

- **Introduction** and **Elimination** rules

\[
\frac{A \quad B}{A \land B} \quad \text{conjI}
\]

\[
\frac{A \lor B \quad A \lor B}{A \lor B} \quad \text{disjI1/2}
\]

\[
\frac{A \rightarrow B}{A \rightarrow B} \quad \text{impl}
\]

\[
\frac{A \land B}{C} \quad \text{conjE}
\]

\[
\frac{A \lor B}{C} \quad \text{disjE}
\]

\[
\frac{A \rightarrow B}{C} \quad \text{impE}
\]
NATURAL DEDUCTION RULES

\[
\frac{A \quad B}{A \land B} \quad \text{conjI}
\]

\[
\frac{A \land B \quad [A; B] \rightarrow C}{C} \quad \text{conjE}
\]

\[
\frac{A \lor B \quad A \lor B}{A \lor B} \quad \text{disjI1/2}
\]

\[
\frac{A \lor B \quad C}{C} \quad \text{disjE}
\]

\[
\frac{A \rightarrow B \quad A}{B} \quad \text{impl}
\]

\[
\frac{A \rightarrow B \quad C}{C} \quad \text{impE}
\]

For each connective (\(\land, \lor, \) etc):

**introduction** and **elimination** rules
For each connective ($\land$, $\lor$, etc):

**introduction** and **elimination** rules
NATURAL DEDUCTION RULES

For each connective (\(\land\), \(\lor\), etc):

**introduction** and **elimination** rules
**Natural Deduction Rules**

\[
\begin{array}{c}
\frac{A \quad B}{A \land B} \quad \text{conjI} \\
\frac{A}{A \lor B} \quad \frac{B}{A \lor B} \quad \text{disjI1/2} \\
\frac{A \rightarrow B}{A \rightarrow B} \quad \text{impl} \\
\end{array}
\]

\[
\begin{array}{c}
\frac{A \land B \quad [A; B] \rightarrow C}{C} \quad \text{conjE} \\
\frac{A \lor B \quad A \rightarrow C \quad B \rightarrow C}{C} \quad \text{disjE} \\
\frac{A \rightarrow B}{C} \quad \text{impE}
\end{array}
\]

For each connective (\(\land, \lor, \text{etc}\)):

**introduction and elimination** rules
For each connective ($\land$, $\lor$, etc):

**introduction** and **elimination** rules
**Proof by Assumption**

**apply** assumption

proves

1. \([B_1; \ldots; B_m] \Rightarrow C\)

by unifying \(C\) with one of the \(B_i\)
apply assumption

proves

1. \([B_1; \ldots; B_m] \Rightarrow C\)

by unifying \(C\) with one of the \(B_i\)

There may be more than one matching \(B_i\) and multiple unifiers.

Backtracking!

Explicit backtracking command: back
**Intro rules** decompose formulae to the right of $\rightarrow$.

**Apply** (rule `<intro-rule>`)
**Intro rules** decompose formulae to the right of $\implies$.

apply (rule `<intro-rule>`)

Intro rule $\left[ A_1; \ldots; A_n \right] \implies A$ means

$\implies$ To prove $A$ it suffices to show $A_1 \ldots A_n$
Intro rules decompose formulae to the right of $\implies$.

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

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

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

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

$\rightarrow$ unify $A$ and $C$

$\rightarrow$ replace $C$ with $n$ new subgoals $A_1 \ldots A_n$
**Elim** rules decompose formulae on the left of $\implies$.

**apply** (erule $<$elim-rule$>$)
**Elim Rules**

*Elim* rules decompose formulae on the left of $\Rightarrow$.

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

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

- If I know $A_1$ and want to prove $A$ it suffices to show $A_2 \ldots A_n$
**Elim** rules decompose formulae on the left of $\rightarrow$.

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

Elim rule $\begin{array}{c} A_1; \ldots; A_n \end{array} \rightarrow A$ means

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

Applying rule $\begin{array}{c} A_1; \ldots; A_n \end{array} \rightarrow A$ to subgoal $C$:

Like **rule** but also

$\rightarrow$ unifies first premise of rule with an assumption

$\rightarrow$ eliminates that assumption
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 \( (A \rightarrow B \rightarrow C) = (A \land B \rightarrow C) \) in Isabelle

→ prove \( \neg(A \land B) \implies \neg A \lor \neg B \) in Isabelle (tricky!)