MORE GENERAL TYPES

A term can have more than one type.

Example:

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

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

Examples:
\[ \text{int} \Rightarrow \text{bool} \subseteq \alpha \Rightarrow \beta \]
\[ \text{bool} \subseteq \beta \Rightarrow \alpha \]
\[ \alpha \Rightarrow \alpha \]

MOST GENERAL TYPES

Fact: each type correct term has a most general type

Formally:
\[ \Gamma \vdash t : \tau \quad \Rightarrow \exists \sigma. \Gamma \vdash t : \sigma \wedge (\forall \sigma'. \Gamma \vdash t : \sigma' \Rightarrow \sigma' \subseteq \sigma) \]

It can be found by executing the typing rules backwards.

\[ \rightarrow \text{ type checking: checking if } \Gamma \vdash t : \tau \text{ for given } \Gamma \text{ and } \tau \]
\[ \rightarrow \text{ type inference: computing } \Gamma \text{ and } \tau \text{ such that } \Gamma \vdash t : \tau \]

Type checking and type inference on \( \lambda \rightarrow \) are decidable.
**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?**

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

(Alan Turing, 1942)

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

\(=_{\beta}\) 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 \(\lambda \rightarrow\)!

How can typed functional languages then Turing complete?

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

\(Y\) is called fix point operator
used for recursion

---

**Types and Terms in Isabelle**

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

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

\(=_{\tau}\) type constructors: construct a new type out of a parameter type.
Example: int list

\(=_{\tau}\) type classes: restrict type variables to a class defined by axioms.
Example: a :: order

\(=_{\tau}\) schematic variables: variables that can be instantiated.
**Type Classes**

- similar to Haskell’s type classes, but with semantic properties
  - `axclass order < ord`
    - `order_ax`: “x ≤ x”
    - `order_trans`: “[x ≤ y; y ≤ z] ⇒ x ≤ z”
  - theorems can be proved in the abstract
    - `lemma order_conj_trans`: “∀ x :: α. [x < y; y < z] ⇒ x < z”
  - can be used for subtyping
    - `axclass linorder < order`
      - `linorder_linear`: “x ≤ y ∨ y ≤ x”
  - can be instantiated
    - `instance nat :: "{order, linorder}"`

**Schematic Variables**

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

- X and Y must be instantiated to apply the rule

**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 σ(s) = σ(t)

In Isabelle:
Find substitution σ on schematic variables such that σ(s) \(\alpha\beta\gamma\sigma\) (t)

Examples:
- `?X ∧ ?Y = αβγ x ∧ x`  \[ [?X ← x, ?Y ← x] \]
- `?P x = αβγ x ∧ x`  \[ [?P ← λx. x ∧ x] \]
- `P (?f x) = αβγ ?Y x`  \[ [?f ← λx. x, ?Y ← P] \]

Higher Order: schematic variables can be functions.

**Higher Order Unification**

- Unification modulo \(\alpha\beta\) (Higher Order Unification) is semi-decidable
- Unification modulo \(\alpha\beta\gamma\) 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 \(\lambda\) normal form where
  - each occurrence of a schematic variable is of the form \(\lambda f \ t_1 \ldots t_n\)
  - the \(t_1 \ldots t_n\) are \(\eta\)-convertible into \(n\) distinct bound variables
WE HAVE LEARNED SO FAR...

- Simply typed lambda calculus: $\lambda^\rightarrow$
- 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

PROOFS IN ISABELLE

General schema:

- \textbf{lemma} name: "<goal>"  
- \textbf{apply} <method>  
- \textbf{apply} <method>  
- ...  
- \textbf{done}

- Sequential application of methods until all \textbf{subgoals} are solved.

PREVIEW: PROOFS IN ISABELLE

Slide 13

Slide 14

THE PROOF STATE

1. $A_1 x_1 \ldots x_p [A_1; \ldots; A_n] \Longrightarrow B$
2. $A_y y_1 \ldots y_q [C_1; \ldots; C_m] \Longrightarrow D$

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

Syntax:

theory MyTh = ImpTh1 + ... + ImpThn;
(declarations, definitions, theorems, proofs, ...)
end

⇒ MyTh: name of theory. Must live in file MyTh.thy
⇒ ImpThi: name of imported theories. Import transitive.

Unless you need something special:

theory MyTh = Main:

NATURAL DEDUCTION RULES

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ⊢ B</td>
<td>A ∧ B</td>
</tr>
<tr>
<td>A ⊢ B</td>
<td>A ∨ B</td>
</tr>
<tr>
<td>A ⊢ B</td>
<td>A → B</td>
</tr>
</tbody>
</table>

For each connective (∧, ∨, etc):
- introduction and elimination rules

Proof by Assumption

apply assumption

proves

1. [B1; ...; Bn] ⊢ C

by unifying C with one of the Bi.

There may be more than one matching Bi and multiple unifiers.

Backtracking!

Explicit backtracking command: back

Intro rules decompose formulae to the right of →→.

apply (rule <intro-rule>)

Intro rule [A1; ...; An] ⊢ A means

⇒ To prove A it suffices to show A1 ... An.

Applying rule [A1; ...; An] ⊢ A to subgoal C:

⇒ unify A and C
⇒ replace C with n new subgoals A1 ... An
Elim rules decompose formulae on the left of $\Rightarrow$.

% apply (erule <elim-rule>)

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

$\Rightarrow$ If I know $A_i$ and want to prove $A$, it suffices to show $A_2; \ldots; A_n$.

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

Like rule but also

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

$\Rightarrow$ eliminates that assumption