Overview of Isabelle/HOL

- HOL = Higher-Order Logic
- HOL = Types + Lambda Calculus + Logic
- HOL has
  - datatypes
  - recursive functions
  - logical operators ($\land$, $\lor$, $\neg$, $\rightarrow$, $\forall$, $\exists$, ...)
- Contains propositional logic, first-order logic
- HOL is very similar to a functional programming language
- Higher-order = functions are values, too!
- We’ll start with propositional and first order logic
Types

Syntax:

\[ \tau ::= (\tau) \]
\[ \ | \ \text{bool} \ | \ \text{nat} \ | \ ... \ \text{base types} \]
\[ \ | \ 'a \ | \ 'b \ | \ ... \ \text{type variables} \]
\[ \ | \ \tau \Rightarrow \tau \ \text{total functions (ascii : =>)} \]
\[ \ | \ \tau \times \tau \ \text{pairs (ascii : *)} \]
\[ \ | \ \tau \ \text{list} \ \text{lists} \]
\[ \ | \ ... \ \text{user-defined types} \]

Parentheses: \[ T1 \Rightarrow T2 \Rightarrow T3 \equiv T1 \Rightarrow (T2 \Rightarrow T3) \]
Terms: Basic syntax

- Syntax:
  
  \[
  \text{term} ::= (\text{term}) \\
  \text{constant or variable (identifier)} \\
  \text{function application} \\
  \text{function “abstraction”} \\
  \text{lots of syntactic sugar}
  \]

- Examples:  
  \[f (g \; x) \; y \quad h (\lambda x. f (g \; x))\]

- Parentheses:  
  \[f \; a_1 \; a_2 \; a_3 \equiv ((f \; a_1) \; a_2) \; a_3\]

- Note: Formulae are terms
λ-calculus in a nutshell

- Informal notation: \( t[x] \)
  term \( t \) with 0 or more free occurrences of \( x \)

- **Function application:**
  \( f \ a \) is the function \( f \) called with argument \( a \).

- **Function abstraction:**
  \( \lambda x. t[x] \) is the function with formal parameter \( x \) and body/result \( t[x] \), i.e. \( x \mapsto t[x] \).
Computation:

Replace formal parameter by actual value

("β-reduction"): \((\lambda x. t[x])a \sim_\beta t[a]\)

Example: \((\lambda x. x + 5) 3 \sim_\beta (3 + 5)\)

Isabelle performs \(\beta\)-reduction automatically

Isabelle considers \((\lambda x. t[x])a\) and \(t[a]\) equivalent
Terms and Types

- **Terms must be well-typed!**

- The argument of every function call must be of the right type

- **Notation:** $t :: \tau$ means $t$ is well-typed term of type $\tau$
Isabelle automatically computes ("infer") the type of each variable in a term.

In the presence of overloaded functions (functions with multiple, unrelated types) not always possible.

User can help with type annotations inside the term.

Example:  \( f(x : \text{nat}) \)
Some predefined syntactic sugar:

- **Infix:** +, −, #, @, ...
- **Mixfix:** if_then_else_, case_of_, ...
- **Binders:** \( \forall x. P x \) means \((\forall)(\lambda x. P x)\)

Prefix binds more strongly than infix:

\[
! f x + y \equiv (f x) + y \not\equiv f (x + y) !
\]

Annotations of definitions let you add your own.
Symbol Translations

<table>
<thead>
<tr>
<th>x-symbol</th>
<th>( \forall )</th>
<th>( \exists )</th>
<th>( \lambda )</th>
<th>( \neg )</th>
<th>( &amp; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii (1)</td>
<td>(&lt;\text{forall}&gt;)</td>
<td>(&lt;\text{exists}&gt;)</td>
<td>(&lt;\text{lambda}&gt;)</td>
<td>(&lt;\text{not}&gt;)</td>
<td>(&lt;\text{and}&gt;)</td>
</tr>
<tr>
<td>ascii (2)</td>
<td>ALL</td>
<td>EX</td>
<td>%</td>
<td>( \sim )</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x-symbol</th>
<th>( \lor )</th>
<th>( \longrightarrow )</th>
<th>( \Rightarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii (1)</td>
<td>(&lt;\text{or}&gt;)</td>
<td>(&lt;\text{longrightarrow}&gt;)</td>
<td>(&lt;\text{Rightarrow}&gt;)</td>
</tr>
<tr>
<td>ascii (2)</td>
<td></td>
<td>( \rightarrow )</td>
<td>( =&gt; )</td>
</tr>
</tbody>
</table>

- (1) is converted to unicode tokens (sometimes with help from you)
- (2) remains as ascii (most of the time)

See Appendix B of Concrete Semantics for more complete list
**Formulae (first Approximation)**

- **Syntax** (in decreasing priority):

  \[
  \text{form} ::= (\text{form}) \quad | \quad \text{term} = \text{term} \\
  | \quad \neg \text{form} \quad | \quad \text{form} \land \text{form} \\
  | \quad \text{form} \lor \text{form} \quad | \quad \text{form} \rightarrow \text{form} \\
  | \quad \forall x. \text{form} \quad | \quad \exists x. \text{form}
  \]

  and some others

- **Scope** of quantifiers: as far to the right as possible
Examples

- \( \neg A \land B \lor C \equiv ((\neg A) \land B) \lor C \)
- \( A \land B = C \equiv A \land (B = C) \)
- \( \forall x. P \, x \land Q \, x \equiv \forall x. (P \, x \land Q \, x) \)
- \( \forall x. \exists y. P \, x \land Q \, x \equiv \forall x. (\exists y. (P \, x \land Q \, x)) \)
Type bool

Formulae = terms of type bool

- True::bool
- False::bool
- ¬ :: bool ⇒ bool
- ∧, ∨, ... :: bool ⇒ bool

Large :

if-and-only-if: =
but binds more tightly
Isabelle supports overloading of constants through type classes
! Numerals and arithmetic operations are overloaded:
0, 1, 2, . . . :: nat or :: int or :: real (or others)
+ :: nat ⇒ nat ⇒ nat and
+ :: real ⇒ real ⇒ real (and others)
Suc only at :: nat ⇒ nat
You need type annotations: 1 :: nat, x + (y :: nat)
. . . unless the context is unambiguous: Suc 0
Basic syntax and constructs

- `[ ]`: empty list
- `x # xs`: list with first element `x` ("head") and rest `xs` ("tail")
- Syntactic sugar: `[x_1, \ldots, x_n] \equiv x_1 # \ldots # x_n # [ ]`

Large library:

hd, tl, map, size, filter, set, nth, take, drop, distinct, ...

Don’t reinvent, reuse!

\sim \text{HOL/List.thy}
When writing terms and types in `.thy` files:

- Types and terms need to be enclosed in double quotes "..."
- Except for single identifiers, e.g. `tmpvar` or `'a`

"..." won’t always be shown on slides
Defining Things
Keywords:

- **typedef**: Primitive for type definitions
  - Must build a model and prove it nonempty
  - Only real way of introducing a new type with new properties
  - probably won’t directly use in this class

- **typedecl**: Pure declaration
  - New type with no properties (except that it is non-empty)
Introducing New Types: typedef

Existing type $ty$

Model for new type $M$

New type $nty$

$Abs_{nty}$

$Rep_{nty}$
Introducing New Types

Keywords:

- **type_synonym**: Abbreviation - used only to make theory files more readable
- **datatype**: Defines recursive data-types; solutions to free algebra specifications
  - Basis for primitive recursive function definitions and patterns
- **record**: introduces a record type scheme, introducing its fields. To be covered later (maybe).
Examples

- `type_synonym name = string`
- `type_synonym ('a,'b)foo = "'a list * 'b"

Type abbreviations are expanded immediately after parsing

Not present in internal representation and Isabelle output
**datatype: The Example**

```plaintext
datatype 'a list = Nil | Cons 'a ''a list
```

- **Properties:**
  - Type constructors: `list` of one argument
  - Term constructors: `Nil :: 'a list`
    ```plaintext
    Cons :: 'a ⇒ 'a list ⇒ 'a list
    ```
  - Distinctness: `Nil ≠ Cons x xs`
  - Injectivity:
    ```plaintext
    (Cons x xs = Cons y ys) = (x = y ∧ xs = ys)
    ```

- Will use for programming language abstract syntax
A Recursive Function: List Append

- Definition by recursion:

  ```
  fun app :: 'a list ⇒ 'a list ⇒ 'a list where
  app [] ys = ___
  | app (x # xs) ys = ___app xs ...____
  ```

- One rule per pattern
- Recursive calls only applied to "smaller" arguments
  - Uses heuristics to find order based on structural orderings and lexicographic orderings
  - Fails if it can’t find an ordering
- Guarantees termination (total function) (or fails)