Overview of Isabelle/HOL

- **HOL =** Higher-Order Logic
- **HOL =** Types + Lambda Calculus + Logic
- **HOL has**
  - datatypes
  - recursive functions
  - logical operators (\(\land, \lor, \neg, \rightarrow, \forall, \exists, \ldots\))
- 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:

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

Parentheses: \[ T_1 \Rightarrow T_2 \Rightarrow T_3 \equiv T_1 \Rightarrow (T_2 \Rightarrow T_3) \]
Terms: Basic syntax

Syntax:

\[
\text{term} ::= (\text{term}) \\
| \ c \ | \ x \quad \text{constant or variable (identifier)} \\
| \ \text{term term} \quad \text{function application} \\
| \ \lambda x. \ \text{term} \quad \text{function “abstraction”} \\
| \ \ldots \quad \text{lots of syntactic sugar}
\]

Examples: \( f (g \ x) \ y \ h (\lambda x. f (g \ x)) \)

Parentheses: \( f \ a_1 \ a_2 \ a_3 \equiv ((f \ a_1) \ a_2) \ a_3 \)

Note: Formulae are terms
\( \text{Informal notation: } t[x] \)

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”): \((\lambda x. t[x])a \leadsto_\beta t[a]\)

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

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>( \land )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii (1)</td>
<td>\textbackslash{forall}</td>
<td>\textbackslash{exists}</td>
<td>\textbackslash{lambda}</td>
<td>\textbackslash{not}</td>
<td>\textbackslash{and}</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>( \rightarrow )</th>
<th>( \Rightarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ascii (1)</td>
<td>\textbackslash{or}</td>
<td>\textbackslash{longrightarrow}</td>
<td>\textbackslash{Rightarrow}</td>
</tr>
<tr>
<td>ascii (2)</td>
<td></td>
<td>\textbackslash{implies}</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):

```
form ::= (form) | term = term
| ¬form     | form ∧ form
| form ∨ form | form → form
| ∀x. form  | ∃x. form
```

and some others

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

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

- $0, 1, 2, \ldots :: \text{nat}$ or $:: \text{int}$ or $:: \text{real}$ (or others)
- $+ :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ and
- $+ :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}$ (and others)
- $\text{Suc}$ only at $:: \text{nat} \Rightarrow \text{nat}$

You need type annotations: $1 :: \text{nat}$, $x + (y :: \text{nat})$

... unless the context is unambiguous: $\text{Suc 0}$
Type **list**

- 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!
  \[ \leadsto \text{HOL/List.thy} \]
Concrete Syntax

- 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
Introducing New Types

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).
**type_synonym**

- **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

datatype 'a list = Nil | Cons 'a "'a list"

- Properties:
  - Type constructors: list of one argument
  - Term constructors: Nil :: 'a list
  - Cons :: 'a ⇒ 'a list ⇒ 'a list
  - Distinctness: Nil ≠ Cons x xs
  - Injectivity:
    
    (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 Nil ys = ___
    app (Cons 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)