Functional Programming in HOL

Martin Hofmann

23.05.2007

Reading Club Isabelle/HOL
Overview

1. Revision
2. Type definition
3. Function definition
4. Putting things together
5. Proofs Methods
6. Useful commands
**System Architecture**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proof General</td>
<td>(X)Emacs based interface</td>
</tr>
<tr>
<td>Isabelle/HOL</td>
<td>Isabelle instance for HOL</td>
</tr>
<tr>
<td>Isabelle</td>
<td>generic theorem prover</td>
</tr>
<tr>
<td>Standard ML</td>
<td>implementation language</td>
</tr>
</tbody>
</table>

\[
\text{HOL} = \text{Functional Programming} + \text{Logic}
\]
Basic constructs

**Implication** \( \Longrightarrow (==>) \)
For separating premises and conclusions of theorems

**Equality** \( \equiv (==) \)
For definitions

**Universal Quantifier** \( \forall (!) \)
Rarely needed

**Attention**
- Do not use inside HOL formulae
- use \( \rightarrow (\rightarrow), =, \forall (!) \) instead
Isabelle’s meta-logic

Notation

### Abbreviation

\([A_1; \ldots; A_n] \Rightarrow B\)

abbreviates

\(A_1 \Rightarrow \ldots \Rightarrow A_n \Rightarrow B\)

; \(\approx\) ‘and’

### Proof state:

\(\forall x_1, \ldots x_n. [A_1; \ldots; A_n] \Rightarrow B\)

- \(x_1, \ldots x_n\) local constants
- \(A_1; \ldots; A_n\) local variables
- \(B\) actual (sub)goal
1 Revision

2 Type definition
   Introducing new types
   Predefined data types

3 Function definition

4 Putting things together

5 Proofs Methods

6 Useful commands
New types

Keywords:

- typedec: pure declaration
- types: abbreviation
- datatype: recursive datatype
**typedec**

*typedec name*

Introduces new “opaque” type *name* without definition

**Example**

*typedec book*
Introducing new types

**types**

```plaintext
 type = τ
```

Introduces an abbreviation *name* for type *τ*

**Example**

```plaintext
 types
 name = string
 ('a,'b)foo = "'a list × 'b list"
```

- expanded immediately after parsing
- not present in internal representation and output
**Introducing new types**

**datatype**

\[
\text{datatype } (\alpha_1, \ldots, \alpha_n) t = C_1 \tau_{11} \ldots \tau_{1k_1} | \ldots | C_m \tau_{m1} \ldots \tau_{mk_m}
\]

**Types:**  \( C_i :: \tau_{i1} \Rightarrow \ldots \Rightarrow \tau_{ik_i} \)

**Distinctness:**  \( C_i \ldots \neq C_j \ldots \) if \( i \neq j \)

**Injectivity:**  \( (C_i x_1 \ldots x_n = C_i y_1 \ldots y_n) = (x_1 = y_1 \land \ldots \land x_n = y_n) \)

**Example**

**datatype**  
\`
'a list = Nil | Cons 'a ""a list"
``

**Distinctness and Injectivity are applied automatically, Induction not (⟲ later)!**
Introducing new types

**datatype**

\[
\text{datatype } (\alpha_1, \ldots, \alpha_n) t = C_1 \tau_{11} \ldots \tau_{1k_1} | \ldots | C_m \tau_{m1} \ldots \tau_{mk_m}
\]

**Types:** \( C_i :: \tau_{i1} \Rightarrow \ldots \Rightarrow \tau_{ik_i} \)

**Distinctness:** \( C_i \ldots \neq C_j \ldots \text{if } i \neq j \)

**Injectivity:** \((C_i x_1 \ldots x_n = C_i y_1 \ldots y_n) = (x_1 = y_1 \land \ldots \land x_n = y_n)\)

**function size**

- defined automatically (overloaded)
- zero for all constructors that do not have an argument of type \( t \)
- one plus sum of the size of all arguments of type \( t \), for all other constructors
Predefined data types

**Natural Numbers**

- behaves as defined as
  
  \[
  \text{datatype nat} = 0 \mid \text{Succ nat}
  \]

- +, -, *, div, mod, min, max, \leq, <, LEAST

- 1 :: nat unfolded automatically (mostly)

**List**

- Lists in `Main.thy`

- with usual `hd, tl, length`
Predefined data types

Built-in data types

Pairs

- \((a_1, a_2) :: \tau_1 \times \tau_2\)
- with \texttt{fst}, \texttt{snd}

Option

- \texttt{datatype 'a option = None | Some 'a}
1 Revision

2 Type definition

3 Function definition
   Definition by Example
   Definition using Recursion

4 Putting things together

5 Proofs Methods

6 Useful commands
New functions

Only total functions

- totality ensures consistency
  \[ f(x) = f(x) + 1 \Rightarrow 0 = 1 \text{ not !} \]
- fixed constructs to introduce data types and functions

Function definition

- Non-recursive with `defs/constdefs`
  → No problem
- Primitive-recursive with `primrec`
  → Terminating by construction
- Well-founded recursion with `recdef`
  → User must (help to) prove termination (⟲ later)
Definition by Example

Definition by example

Declaration

cconsts

  sq :: "nat ⇒ nat"

Definition

defs

  sq_def: "sq n ≡ n*n"

Declaration + definition

constdefs

  sq :: "nat ⇒ nat"

  "sq_def: sq n ≡ n*n"
Watch Out!

Pitfalls with definitions

constdefs
prime :: "nat ⇒ bool"
"prime p ≡ 
\(1 < p \land (\forall m. m \text{ dvd } p \rightarrow m = 1 \lor m = p)\)"

- Not a definition: free \(m\) not on left-hand side
- Every free variable on the rhs must occur on the lhs

"prime p ≡ 
\(1 < p \land (\forall m. m \text{ dvd } p \rightarrow m = 1 \lor m = p)\)"
Definition using Recursion

**Primitive Recursion**

- `primrec` followed by list of equation
- \( f \ x_1 \ldots (C_1 \ y_1 \ldots y_k) \ldots x_n = r \)
- \( r \) must be structurally smaller, i.e. of the form
  \( f \ldots y_i \ldots \) for some \( i \)

```
primrec
  "f 0 = ..."
  "f(Succ n) = ... f n ..."
```
Case Distinction

if then else

conditionals in common mixfix

case

- every datatype introduces a `case` construct
- `case xs of [[] ⇒ ... | x xs ⇒ ... x ... xs ...`
- all constructors must be present, order fixed
- no nested patterns but nested `case`-expressions
- use ( ) to indicate scope
<table>
<thead>
<tr>
<th></th>
<th>Revision</th>
<th>Type definition</th>
<th>Function definition</th>
<th>Putting things together</th>
<th>Proofs Methods</th>
<th>Useful commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Revision</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Type definition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Function definition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><strong>Putting things together</strong></td>
<td><strong>Theories and Proofs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Proofs Methods</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Useful commands</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Theory = Module

Syntax

theory My Th
imports Th₁,..., Thₙ
begin
  (declarations, definitions, theorems, proofs)
end

MyTh name of theory, must live in file MyTh.thy

Thₓ imported theories, import transitive

Usually theory MyTh imports Main
Theories and Proofs

Proofs

General schema

lemma name: "...
apply (...)
apply (...)
...
done

- types and formulae need to be inclosed in "...
- except single identifiers, e.g. 'a
1. Revision

2. Type definition

3. Function definition

4. Putting things together

5. Proofs Methods
   - Basic Methods
   - Simplification
   - The simp Method
   - Assumptions
   - Definitions
### Basic Methods

#### Proof Methods I

**Structural Induction**
- `(induct_tac x)`, where `x` is a free variable with type of datatype in first subgoal
- generates one new subgoal per constructor

**Case Distinction**
- `(case_tac x)
- generates one new subgoal per constructor
- weaker then `induct_tac`
Proof Methods II

Simplification and a bit of logic

- (auto)
- tries to solve as many subgoals as possible
- uses simplification `simp` and basic logical reasoning
- ignores quantified formulae, logical connectives and all operation apart from addition

`arith`

- more general than `auto` or `simp`
- attempts to prove the first subgoal
- only quantifier-free linear arithmetic formulae, logical connectives included
Simplification

- simplification as term rewriting
- repeatedly use equations from left to right
- used in `auto` and `simp`

Declaration

- for default rules: `lemma xyz [simp] "..."`
- use selectively: `declare xyz [simp] or [simp_del]`

Simplification can run forever! It is your responsibility to ensure termination!
The simp Method

simp \textit{list of modifiers}

- uses all theorems declared as [simp]
- simplifies the first subgoals
- simp\_all simplifies all
- fine tuning with modifiers:
  - add: \textit{list of theorem names}
  - del: \textit{list of theorem names}
  - only: \textit{list of theorem names}

\texttt{apply\( (simp\ add:\ mod\_mult\_distrib\ add\_mult\_distrib)\)}
Simplification of/with Assumptions

Toggle use of assumptions

- assumptions are part of simplification process
- used as rules and simplified themselves
- may lead to nontermination
- use modifiers:
  - no_asm
    - ignored completely
  - no_asm_simp
    - not simplified, but used for simplification
  - no_asm_use
    - simplified, but not used in simplification of each other or the conclusion
Definitions

Rewriting with Definitions

- constant definitions are not simplified automatically
- manually with `apply(simp only: xyz_def)`
- or apply `(unfold xyz_def) !` acts on all subgoals!

- `f x y ≡ t` can only unfold occurrences with at least two arguments
- `f ≡ λx y. t` unfolds all

- every construct has its definition `let_def, xor_def, not_def...`
- `<name>_def ⊇` see reference of your favourite logic (e.g. `logic-HOL.pdf`)
Splitting

**split**

- goal contains case- or if-expression
- proof by case distinction
- `split_if` splits boolean condition
- not for case
- but every datatype `t` comes with `t.split`

**t.split_asm**

split if or case-expressions in assumptions

**t.splits**

combines `t.split` and `t.split_asm`
1. Revision
2. Type definition
3. Function definition
4. Putting things together
5. Proofs Methods
6. Useful commands
**Commands**

- **undo**: undo last step
- **redo**: redo last undo
- **defer**: move first subgoal back
- **prefer n**: move $n^{th}$ subgoal to the front
- **done**: finish proof
- **oops**: abandon proof attempt
- **sorry**: “complete” proof, for top down, assume first, proof later
- **pr**: show current proof state
- **thm name**: print theorem *name*
- **kill**: abandon current theory
ML commands

ML "..."
set flag_name set flag to true
reset flag_name set flag to false
some flag names:
show_types
show_brackets
trace_simp show simplification trace