

Isabelle Tutorial: HOL and its Specification Constructs

Burkhart Wolff

Université Paris-Sud

What we will talk about

Isabelle with:

- its System Framework
- the Logical Framework
- the Isabelle/HOL Environment
- Proof Contexts and Structured Proof
- Tactic Proofs (“apply style”)

Introduction to Isabelle/HOL

Basic HOL Syntax

- HOL (= Higher-Order Logic) goes back to Alonzo Church who invented this in the 30ies ...
- “Classical” Logic over the λ -calculus with Curry-style typing (in contrast to Coq)
- Logical type: “bool” injects to “prop”. i.e

Trueprop :: “bool \Rightarrow prop”

is wrapped around any HOL-Term without being printed:

Trueprop A \Rightarrow Trueprop B is printed: A \Rightarrow B but A::bool!

Basic HOL Syntax

- Logical connective syntax (Unicode + ASCII):

input:	print:	alt-ascii input
--------	--------	-----------------

– “ <code>_ \<and> _</code> ”	“ <code>_ ^ _</code> ”	“ <code>_ & _</code> ”
– “ <code>_ \<or> _</code> ”	“ <code>_ v _</code> ”	“ <code>_ _</code> ”
– “ <code>_ \<implies> _</code> ”	“ <code>_ -> _</code> ”	“ <code>_ --> _</code> ”
– “ <code>_ \<not> _</code> ”	“ <code>_ ~ _</code> ”	“ <code>_ - _</code> ”
– “ <code>\<forall> x. P</code> ”	“ <code>\forall x. P</code> ”	“ <code>! x. P x</code> ”
– “ <code>\<exists> x. P</code> ”	“ <code>\exists x. P</code> ”	“ <code>? x. P x</code> ”

Basic HOL Rules

- Some (almost) basic rules in HOL

$$\frac{Q}{\neg\neg Q}$$

$$\frac{\neg\neg Q}{Q} \text{notnotE}$$

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B} \text{impI} \quad \frac{A \rightarrow B \quad A}{B} \text{mp}$$

$$\frac{A}{A \vee B} \text{disjI1}$$

$$\frac{B}{A \vee B} \text{disjI2}$$

$$\frac{\begin{array}{ccc} [A] & & [B] \\ \vdots & & \vdots \\ A \vee B & Q & Q \end{array}}{Q} \text{disjE}$$

Basic HOL Rules

- Some (almost) basic rules in HOL

$$\frac{\begin{array}{c} [A, B] \\ \vdots \\ Q \end{array}}{Q} \text{conjE} \qquad \frac{A \quad B}{A \wedge B} \text{conjI}$$

Basic HOL Rules

- HOL is an equational logic, i.e. a system with the constant “`_=_`::'a 'a bool” and the rules:

$$\frac{}{x = x} \text{ refl}$$

$$\frac{s = t}{t = s} \text{ sym}$$

$$\frac{r = s \quad s = t}{r = t} \text{ trans}$$

$$\frac{\wedge x. \ s \ x = t \ x}{s = t} \text{ ext}$$

$$\frac{s = t \quad P \ s}{P \ t} \text{ subst}$$

Typed Set-theory in HOL

- The HOL Logic comes immediately with a typed set - theory: The type

$$\alpha \text{ set } \equiv \alpha \Rightarrow \text{bool}, \text{ that's it !}$$

can be defined isomorphically to its type of characteristic functions !

- **THIS GIVES RISE TO A RICH SET THEORY DEVELOPED IN THE LIBRARY (Set.thy).**

Typed Set Theory: Syntax

- Logical connective syntax (Unicode + ASCII):

input:

“_ \<in> _”

“{. }”

“_ \<union> _”

“_ \<inter> _”

“_\<subseteqq>_”

... .

print:

“_ ∈ _”

{x. True}

“_ ∪ _”

“_ ∩ _”

“_ ⊆ _”

alt-ascii input

“_ : _”

{x. True} $\wedge x = x\}$ for example

“_ Un _”

“_ Int _”

“_ <= _”

Inspection Commands

- Type-checking terms:

term “<hol-term>”

example: term “(a::nat) + b = b + a”

- Evaluating terms:

value “<hol-term>”

1

example: term “(3::nat) + 4 = 7”

Exercise demo3.thy

- make yourself familiar with syntax of types
write types and terms in HOL.
- make yourself familiar with the HOL library.
search for HOL-thm's containing specific logical connectives.
- State for example:

$$A \Rightarrow B \Rightarrow C \Rightarrow (A \wedge B) \wedge C \quad (* \llbracket A; B; C \rrbracket \Rightarrow (A \wedge B) \wedge C *)$$

$$P \rightarrow P \vee (Q \wedge R) \quad (* \text{ we ignore trivials like } P \Rightarrow P *)$$

$$P \rightarrow Q \vee (P \wedge \neg Q)$$

$$P \vee \neg P$$

- State some set-theoretic lemmas.

Specification Commands

- Simple Definitions (Non-Rec. core variant):

```
definition f::"<τ>"  
  where <name> : "f x1 ... xn = <t>"
```

example: definition C::"bool ⇒ bool"

where "C x = x"

- Type Definitions:

```
typedef ('a1..'an) κ =  
  "<set-expr>" <proof>
```

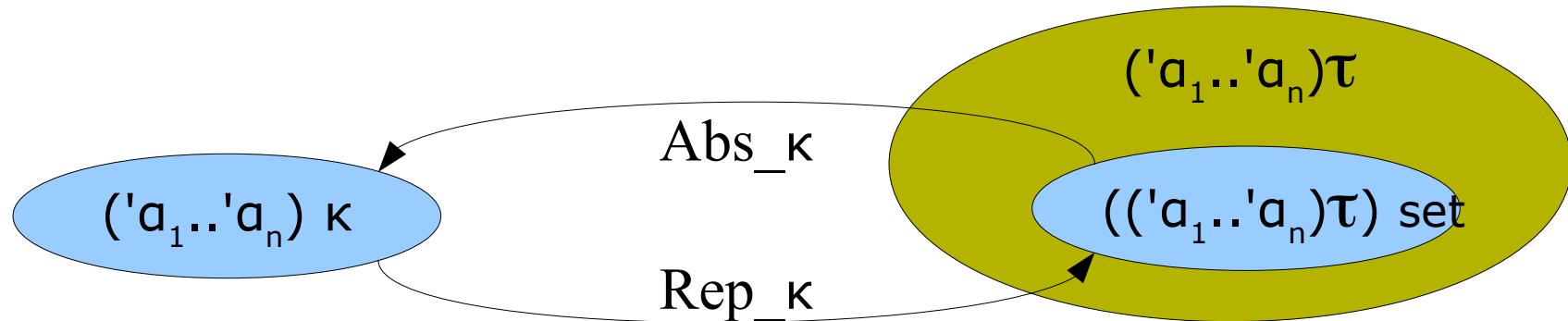
example: typedef even = "{x::int. x mod 2 = 0}

Semantics of a „Type Definition“

- Idea: Similar to constant definitions; we define the new entity (“a type”) by an old one.
- For Type Definitions, we define the new type to be isomorphic to a (non-empty) subset of an old one.
- The Isomorphism is stated by three (conservative) axioms.

Semantics of a „Type Definition“

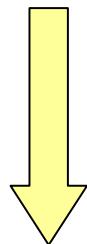
- Idea: Similar to constant definitions; we define the new entity ("a type") by an old one.



Isabelle Specification Constructs

- Type definition:

$(\Sigma, A) \in \Theta$



```
typedef ('a1..'an) κ =  
  "<expr:: (('a1..'an)τ) set>" <proof>
```

$(\Sigma + ('a₁..'a_n)κ + Abs_κ::('a₁..'a_n)τ ⇒ ('a₁..'a_n)κ$

$+ Rep_κ::('a₁..'a_n)κ ⇒ ('a₁..'a_n)τ$

$A + \{Rep_κ_inverse \mapsto Abs_κ (Rep_κ x) = x\}$

$+ \{Rep_κ_inject \mapsto (Rep_κ x = Rep_κ y) = (x = y)\}$

$+ \{Rep_κ \mapsto Rep_κ x \in \{x. expr x\}\} \in \Theta'$

- where the type-constructor κ is “fresh” in Θ
- $expr$ is closed
- $<expr:: ('a₁..'a_n)τ set>$ is non-empty (to be proven by a witness)

Isabelle Specification Constructs

- Major example:
The introduction of the cartesian product:

subsubsection {* Type definition *}

definition Pair_Rep :: "'a ⇒ 'b ⇒ 'a ⇒ 'b ⇒ bool"

where "Pair_Rep a b = ($\lambda x y. x = a \wedge y = b$)"

definition "prod = {f. $\exists a b. f = \text{Pair_Rep } (a :: 'a) (b :: 'b)}$ "

typedef ('a, 'b) prod (infixr "*" 20) = "prod :: ('a ⇒ 'b ⇒ bool) set"

unfolding prod_def by auto

type_notation (xsymbols) "prod" ("(_ ×/ _)") [21, 20] 20)

Specification Mechanism Commands

- Datatype Definitions (similar SML):
(Machinery behind : complex series of const and typedefs !)

```
datatype ('a1..'an) Θ =  
    <c> :: "<τ>" | ... | <c> :: "<τ>"
```

- Recursive Function Definitions:
(Machinery behind: Veeeery complex series of const and typedefs and automated proofs!)

```
fun <c> :: "<τ>" where  
    "<c> <pattern> = <t>"  
    | ...  
    | "<c> <pattern> = <t>"
```

Specification Mechanism Commands

- Datatype Definitions (similar SML):
(Machinery behind : complex !)

```
datatype ('a1..'an) Θ ≡  
  <c> :: "<τ>" | . . . | <c> :: "<τ>"
```

- Recursive Function Definitions.
(Machinery behind: Veeeery complex!)

```
fun <c> :: "<τ>" where  
  "⟨c⟩ <pattern> = <t>"  
  | ...  
  | "⟨c⟩ <pattern> = <t>"
```

Specification Mechanism Commands

- Inductively Defined Sets:

```
inductive <c> [ for <v>:: "< $\tau$ >" ]  
  where <thmname> : "< $\phi$ >"  
        | ...  
        | <thmname> = < $\phi$ >
```

example: inductive path for rel :: "a \Rightarrow 'a \Rightarrow bool"
where base : "path rel x x"
| step : "rel x y \Rightarrow path rel y z \Rightarrow path rel x z"

Specification Mechanism Commands

- Inductively Defined Sets:

```
inductive <c> [ for <v>:: "< $\tau$ >" ]  
where <thmname> : "< $\phi$ >"  
| ...  
| <thmname> = < $\phi$ >
```

example: inductive path for rel :: "a \Rightarrow 'a \Rightarrow bool"
where ~~i.e.~~ base : "path rel x x"

| step : "rel x y \Rightarrow path rel y z \Rightarrow path rel x z"

NOTE: Isabelle HOL compiles this internally to axiomatic "model" in HOL!!

Specification Mechanism Commands

- Extended Notation for Cartesian Products: records
(as in SML or OCaml; gives a slightly OO-flavor)

```
record    <c> = [<record> + ]
  tag1 :: "<τ1>"
  ...
  tagn :: "<τn>"
```

- ... introduces also semantics and syntax for
 - **selectors** : tag₁ x
 - **constructors** : (tag₁ = x₁, ... , tag_n = x_n)
 - **update-functions** : x (tag₁ := x_n)

Tools: The Code-Generator

- Isabelle also generates to each data- and function definition SML Code.
- The latter is accessible, in a complied structure, or as short-hand, via anti-quotations in ML code:

```
ML{* val rev = @{code reverse};  
     rev Isabelle.Generated_Code.Seq  
           (2, Isabelle.Generated_Code.Empty);  
     *}
```

Screenshot with Examples

The screenshot shows the Isabelle/Isar IDE interface. The main window displays a theory file named Seq.thy containing definitions for sequences and functions like conc and reverse. A code completion tooltip is open over the 'conc' function definition, showing the type "'a seq ⇒ 'a seq". The right-hand panel shows a tree view of the theory structure, with the 'conc' definition highlighted. The bottom status bar indicates the session name (isabelle, sidekick, UTF-8-Isabelle) and memory usage (84/154Mb).

```
File Edit Search Markers Folding View Utilities Macros Plugins Help
Seq.thy (~/Papers/isar-book/Orsay/WWW/)

imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys" type
| "conc (Seq x xs) ys = Seq x (conc xs ys)"

fun reverse :: "'a seq ⇒ 'a seq"
where
  "reverse Empty = Empty"
| "reverse (Seq x xs) = conc (reverse xs) (Seq x Empty)"

constants
  conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
Found termination order: "(λp. size (fst p)) <*mlex*> {}"
```

Filter: Seq.thy

theory Seq

header {* Finite sequences *}

theory Seq

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"

fun reverse :: "'a seq ⇒ 'a seq"

lemma conc_empty: "conc xs Empty = xs"

lemma conc_assoc: "conc (conc xs ys) zs = conc xs (conc ys zs)"

lemma reverse_conc: "reverse (conc xs ys) = conc (reverse xs) (reverse ys)"

lemma reverse_reverse: "reverse (reverse xs) = xs"

end

100% ▾ Tracing Auto update Update

Output Prover Session

10,6 (149/731)

(isabelle,sidekick,UTF-8-Isabelle) - - - UG 84/154Mb 9:57 PM

Exercise demo3.thy

- Define your own sequence theory with data type and function definitions such as conc.
- Use the code generator.
- Use the simplifier for establishing elementary expressions on Sequences.