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 ⇒ prop"

is wrapped around any HOL-Term without being printed:

Trueprop A $\Longrightarrow$ Trueprop B is printed: A $\Longrightarrow$ B but A::bool!

Basic HOL Syntax

- Logical connective syntax (Unicode + ASCII):
  input: print: alt-ascii input
  - “_ \<and> _” “_∧_” “_ & _”
  - “_ \<or> _” “_ ∨_” “_ | _”
  - “_ \<implies> _” “_ → _” “_ --> _”
  - “_ \<not> _” “¬_" “~ _”
  - “\<forall> x. P“ “∀x. P” “! x. P x”
  - “\<exists> x. P“ “∃x. P” “? x. P x”

Basic HOL Rules

- Some (almost) basic rules in HOL

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

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

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

$$\frac{A \rightarrow B \quad A}{B}\,\text{mp}$$

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

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

$$\frac{A \vee B \quad \begin{array}{c}[A]\\ \vdots\\ Q\end{array} \quad \begin{array}{c}[B]\\ \vdots\\ Q\end{array}}{Q}\,\text{disjE}$$

Basic HOL Rules

- Some (almost) basic rules in HOL

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

Basic HOL Rules

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

$$\frac{}{x = x}\ \text{refl} \qquad \frac{s = t}{t = s}\ \text{sym} \qquad \frac{r = s \quad s = t}{r = t}\ \text{trans}$$

$$\frac{\bigwedge x.\ s\ x = t\ x}{s = t}\ \text{ext} \qquad \frac{s = t \quad P\ s}{P\ t}\ \text{subst}$$

Typed Set-theory in HOL

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

  α set $\cong$ $\alpha \Rightarrow$ bool, that's it !

  can be defined isomorphically to its type of characteristic functions !

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

Typed Set Theory: Syntax

- Logical connective syntax (Unicode + ASCII):

input:	print:	alt-ascii input
“_ \<in> _”	“_ ∈ _”	“_ : _”
“{_ . _}”	*{x. True ∧ x = x}*	*for example*
“_ \<union> _”	“_∪_”	“_ Un _”
“_ \<inter> _”	“_ ∩_”	“_ Int _”
“_\<subseteq>_”	“_ ⊆ _”	“_ <= _”
. . .		

Inspection Commands

- Type-checking terms:

term “<hol-term>”

example: term “(a::nat) + b = b + a"

- Evaluating terms:

value “<hol-term>”

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:

  $A \Longrightarrow B \Longrightarrow C \Longrightarrow (A \wedge B) \wedge C$ (* $[\![A; B; C]\!] \Longrightarrow (A \wedge B) \wedge C$ **)**

  $P \longrightarrow P \vee (Q \wedge R)$ **(*** we ignore trivials like $P \Longrightarrow P$ *)

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

  $P \vee \neg P$

- State some set-theoretic lemmas.

Specification Commands

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

```
definition f::"<τ>"
  where <name> : "f x_1 ... x_n = <t>"
```

example: definition C::"bool ⇒ bool"

where "C x = x"

- Type Definitions:

```
typedef ('α_1..'α_n) κ =
  "<set-expr>" <proof>
```

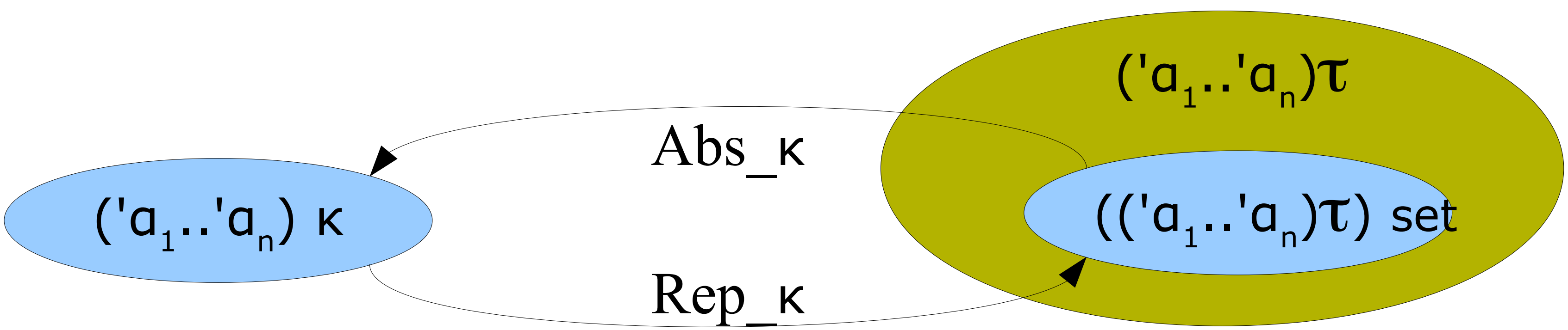
example: typedef even = "{x::int. x mod 2 = 0}

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:

$$(\Sigma, A) \text{"}\in\text{"}\ \Theta$$

typedef $('a_1..'a_n)\ \kappa$ =

"<expr:: $(('a_1..'a_n)\tau)$ set>" <proof>

$$(\Sigma \ + \ ('a_1..'a_n)\ \kappa \ + \ Abs\_\kappa :: ('a_1..'a_n)\tau \Rightarrow ('a_1..'a_n)\kappa$$
$$+ \ Rep\_\kappa :: ('a_1..'a_n)\kappa \Rightarrow ('a_1..'a_n)\tau$$
$$A \ + \ \{Rep\_\kappa\_inverse \mapsto Abs\_\kappa\ (Rep\_\kappa\ x) = x\}$$
$$+ \ \{Rep\_\kappa\_inject \mapsto (Rep\_\kappa\ x = Rep\_\kappa\ y) = (x = y)\}$$
$$+ \ \{Rep\_\kappa \mapsto Rep\_\kappa\ x \in \{x.\ expr\ x\}) \text{"}\in\text{"}\ \Theta'$$

- where the type-constructor κ is "fresh" in Θ
- expr is closed
- <expr:: $('a_1..'a_n)\tau$ 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 = (λx y. x = a ∧ y = b)"


definition "prod = {f. ∃ a b. f = 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: Veeery 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 ('a_1..'a_n) Θ =
 <c> :: "<τ>" | ... | <c> :: "<τ>"
```

- Recursive Function Definitions:
  (Machinery behind: Veeery complex!)

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

NOTE: Isabelle HOL compiles this internally to axiomatic definitions, i.e. a 'model' in HOL!!!

Specification Mechanism Commands

- Inductively Defined Sets:

```
inductive     <c> [ for <v>:: “<τ>” ]
  where  <thmname> : “<ϕ>”
           | ...
           | <thmname> = <ϕ>
```

```
example: inductive path for rel ::"'a ⇒ 'a ⇒ bool"
         where  base : “path rel x x”
              |    step : “rel x y ⟹ path rel y z ⟹ path rel x z”
```

Specification Mechanism Commands

- Inductively Defined Sets:

```
inductive    <c> [ for <v>:: "<τ>" ]
  where  <thmname> : "<ϕ>"
         | ...
         | <thmname> = <ϕ>
```

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

NOTE: Isabelle HOL compiles this internally to axiomatic definitions, i.e. a "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>"
```

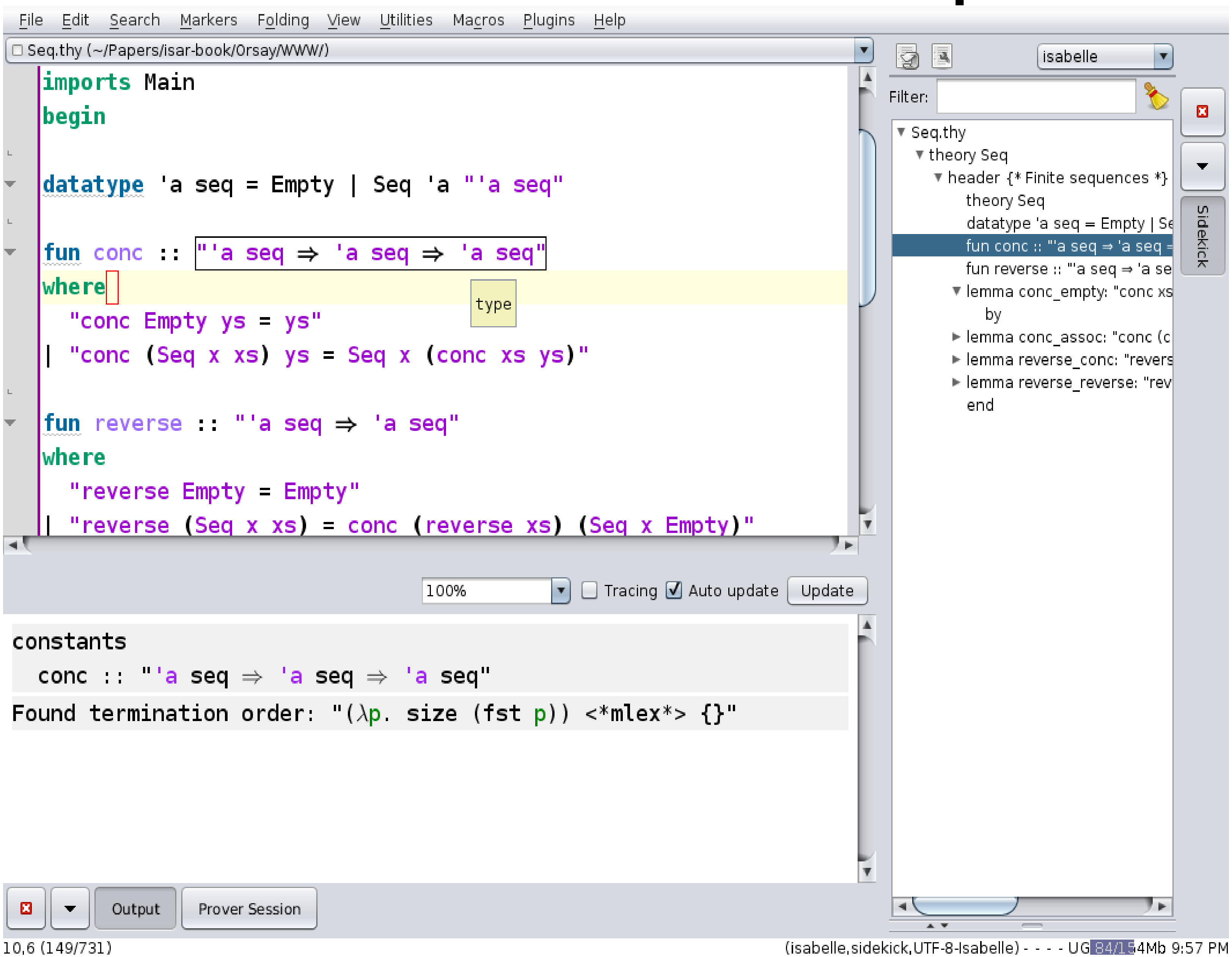
- … introduces also semantics and syntax for
  - selectors : $tag_1\ x$
  - constructors : $⦇\ tag_1 = x_1, \ldots ,\ tag_n = x_n\ ⦈$
  - update-functions : $x\ ⦇\ tag_1 := x_n\ ⦈$

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



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.