Semantics
of Programming Languages (2)

Christian Urban

http://www4.in.tum.de/~urbanc/Teaching/semantics08.html
Any Problems with HW?

- Installation? (I like to know about your experience / How long did you need?)
- Emacs and X-Symbols ($\times$ vs $\times$)
- Interface - stepping through a theory
Any Problems with HW?

- Installation? (I like to know about your experience / How long did you need?)
- Emacs and X-Symbols (\textbackslash <times> vs \times)
- Interface - stepping through a theory

Today: Function definitions and some simple proofs.
For what we are going to do in this course, we import either the theory \texttt{Main} or (later) \texttt{Nominal}.

Except today where we use \texttt{PreList} in order to avoid conflicts with the built-in lists.
Every constant in Isabelle/HOL needs to have a type.

There are many reasons for this: one is to prevent nonsense statements like $1 = \text{true}$.

Two predefined types are booleans and nats:

```
typ "bool"
typ "nat"
```

(typ "boolean" will give an error, since it is not defined)
Types can be constructed using type constructors (e.g. set, list, $\times$, $\Rightarrow$).

\begin{align*}
\text{typ } & \text{"bool set"} \\
\text{typ } & \text{"nat set"} \\
\text{typ } & \text{"nat } \times \text{ bool"} \\
\text{typ } & \text{"nat } \Rightarrow \text{ bool"} \\
\text{typ } & \text{"bool } \Rightarrow \text{ bool } \Rightarrow \text{ nat"}
\end{align*}

Note that $\times$ and $\Rightarrow$ are infix; the default is postfix (from ML).

What do these types signify?
Type Variables

- Polymorphic types are represented using type variables.

\[
\begin{align*}
\texttt{typ "a"} \\
\texttt{typ "b"} \\
\texttt{typ "c ⇒ 'd"}
\end{align*}
\]

- Note the ' in front of type variables.

- A function that takes a set of any type and returns a bool has type:

\[
\texttt{typ "'a set ⇒ bool"}
\]
Every term in Isabelle has a type. For example True and False have type bool.

```
term "True::bool"
term "False::bool"
```

Note that by using :: we can indicate the type of a term (term "True" also works).

```
term "true::bool"
term "false::bool"
```

This is interpreted as two variables of type bool (note the blue colour in the output).
Terms (2)

Try

term "true"
term "false"
term "True"
term "False"

and explain the output.

in 1 and 2 the terms have polymorphic types (since they are variables)
in 3 and 4 both terms have bool as type (they are constants)
New Types

New types can be introduced using the command `datatype`.

```plaintext
datatype 'a list =
    Nil ("[]")
  | Cons "a" "a list" ("_ # _" [101,100] 100)
```

The annotations allow us to use some fancier syntax (respecting precedences).

```plaintext
term "[]"
term "x#xs"
```
Now we have

\[
\text{typ } \text{"nat list"} \\
\text{typ } \text{"bool list"} \\
\text{typ } \text{"a list"}
\]

Note the type of the term constructors.

\[
\text{term } \text{"Nil"} \\
\text{term } \text{"Cons"}
\]

If we want \([\,]\) to be of a particular type, then we have to write e.g. \([\,]\)::\text{nat list}. 
Natural numbers can be defined as a datatype

\[
\text{datatype mynat = MyZero | MySuc "mynat"}
\]

The built-in machinery lets you write (not the one above):

\[
\begin{align*}
\text{term } "0::nat" \\
\text{term } "1::nat" \\
\text{term } "2::nat" \\
\text{term } "\text{(Suc 0)}" \\
\text{term } "\text{Suc (Suc 0)}"
\end{align*}
\]
A Function: Length of Lists

fun
  length :: "a list ⇒ nat"
where
  "length [] = 0"
| "length (x # xs) = 1 + length xs"

We can calculate the result of this function using the command normal_form.

normal_form "length []"
normal_form "length (1 # 2 # 3 # [])"
fun
  set :: "'a list ⇒ 'a set"
where
  "set [] = {}"
| "set (x#xs) = {x}∪(set xs)"

An aside: possible sets are

term "{}"
term "\{1,2,3\} \cap \{2,3,4\}"
term "\{\text{True}\} \cup \{\text{True, False}\}"
term "\{m. \exists n. m = 2 * n\}"
Append Function

fun

append :: "'a list ⇒ 'a list ⇒ 'a list" ("_ @ _")

where

append Nil: "[] @ ys = ys"
append Cons: "(x#xs) @ ys = x#(xs @ ys)"

The syntax annotation allows us to write xs @ ys.

Also note that we labelled each equation (we can use the labels to refer to the equations later on).
List Reversal

\textbf{fun}
\begin{verbatim}
  rev :: \texttt{"'a list \Rightarrow 'a list"}
where
  "rev [] = []"
| "rev (x#xs) = (rev xs) @ (x#[])"
\end{verbatim}

\begin{itemize}
  \item The \texttt{x#[]} can be improved using an abbreviation.
\end{itemize}

\textbf{abbreviation}
\begin{verbatim}
  singleton :: \texttt{"'a \Rightarrow 'a list" ("[\_]\")}
where
  "[x] \equiv x#[]"
\end{verbatim}
fun
   rev :: "'a list ⇒ 'a list"
where
   "rev [] = []"
| "rev (x#xs) = (rev xs) @ [x]"

The x#[] can be improved using an abbreviation.

abbreviation
   singleton :: "'a ⇒ 'a list" ("[-]")
where
   "[x] ≡ x#[]"
fun
  rev :: "'a list ⇒ 'a list"
where
  "rev [] = []"
| "rev (x#xs) = (rev xs) @ [x]"

The x#[] can be improved using an abbreviation.

abbreviation
  singleton :: 'a ⇒ 'a list ("[" x "]")
where
  "[x] = x#[]"

Clever hacks in Isabelle allow you to write for lists:

  []   [True]   [2,3]   [3,2,1]…
Theorems and Lemmas

**theorem** theorem_name:

assumes "assm1"
and "assm2"

... shows "statement"

... The assumptions and the goal must be terms of type bool.

Whether you use **theorem** or **lemma** does not matter.
Theorems and Lemmas

```
theorem theorem_name:
  assumes "assm1"
  and "assm2"
  ...
  shows "statement"
  ...
```

- The assumptions are optional (they also can have labels).

```
theorem theorem_name:
  shows "lhs = rhs"
  ...
```
An Example

Lemma test:
shows “length (xs@ys) = length xs + length ys”
sorry

Sorry is here only for experimentation.

Lemmas can be queried using thm

Thm test
An Example

lemma test:
  shows "length (xs@ys) = length xs + length ys"

sorry

- sorry is here only for experimentation.
- lemmas can be queried using thm

thm test

length (?xs @ ?ys) = length ?xs + length ?ys

We proved something like ∀ xs ys. length (xs @ ys) = length xs + length ys. We just we omit the outermost quantifiers.
lemma test:
  shows "length (xs@ys) = length xs + length ys"

sorry

- sorry is here only for experimentation.
- lemmas can be queried using thm

thm test[where xs="[]" and ys="[True]"]

length ([] @ [True]) = length [] + length [True]

We proved something like \( \forall \, xs \, ys. \, \text{length} (xs \, @ \, ys) = \text{length} \, xs + \text{length} \, ys \). We just we omit the outermost quantifiers.
Proofs

We declared lists to be:

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

With this declaration comes the following induction principle:

```
thm list.induct
```
Induction Principle

The induction principle for lists:

\[ \text{thm list.induct}[\text{no_vars}] \]

\[ [P \; \text{[]}; \; \forall a \; \text{list.} \; P \; \text{list} \implies P \; (a \; \# \; \text{list})] \implies P \; \text{list} \]

1.) \( P \; \text{[]} \)

2.) \( \forall x \; xs. \; P \; xs \implies P \; (x \# xs) \)

\[ \forall xs. \; P \; xs \]
Induction Principle

The induction principle for lists:

\[ \text{thm list.induct[no_vars]} \]
\[ [P []; \forall a \text{ list}. P \text{ list} \implies P (a \# \text{ list})] \implies P \text{ list} \]

1.) \( P [] \)

2.) \( \forall x \text{ xs}. P \text{ xs} \implies P (x \# \text{ xs}) \)

\[ \forall x \text{ xs}. P \text{ xs} \]

If we want to establish a property \( P \) for all lists \( \text{xs} \), we need to show that \( P [] \) holds and that \( P (x \# \text{xs}) \) holds for all \( x \) and \( \text{xs} \), assuming \( P \text{ xs} \) already holds.
Proof by Induction

lemma length_append_test:
  shows "length (xs @ ys) = length xs + length ys"
proof (induct xs rule: list.induct)
  case Nil
  show "length ([] @ ys) = length [] + length ys"
  ...
next
  case (Cons x xs)
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
  ...
qed

You can get the cases from the menu.
An alternative is to write show ?case.
Proof by Induction

lemma length_append_test:
    shows "length (xs @ ys) = length xs + length ys"
proof (induct xs rule: list.induct)
  case Nil
  show "length ([] @ ys) = length [] + length ys"
  
  next
  case (Cons x xs)
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
  
qed

Every show-statement needs a justification.
Proof by Induction

lemma length_append_test:
  shows "length (xs @ ys) = length xs + length ys"
proof (induct xs rule: list.induct)
  case Nil
  show "length ([] @ ys) = length [] + length ys"
  sorry
next
  case (Cons x xs)
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
  sorry
qed

sorry accepts any statement (even false ones!)
Proof by Induction

lemma length_append_test:
  shows "length (xs @ ys) = length xs + length ys"
proof (induct xs rule: list.induct)
case Nil
  show "length ([] @ ys) = length [] + length ys"
  
next
case (Cons x xs)
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
  
qed

We know by definition
  • [] @ ys = ys
  • length [] = 0
Proof by Induction

lemma length_append_test:
  shows "length (xs @ ys) = length xs + length ys"
proof (induct xs rule: list.induct)
  case Nil
  show "length ([] @ ys) = length [] + length ys"
    by simp
next
  case (Cons x xs)
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
    ...
qed

We know by definition
  • [] @ ys = ys
  • length [] = 0
Proof by Induction

lemma length_append_test:
  shows "length (xs @ ys) = length xs + length ys"
proof (induct xs rule: list.induct)
  case Nil
  show "length ([] @ ys) = length [] + length ys"
    by simp
next
  case (Cons x xs)
  have ih: "length (xs@ys) = length xs + length ys" by fact
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
    ...
qed

In this case we have an induction hypothesis.
Proof by Induction

We know by definition

- \((x \# xs) @ ys = x \# xs @ ys\)
- \(\text{length } (x \# xs) = 1 + \text{length } xs\)

(remember for all \(x\) and \(xs\))
Proof by Induction

lemma length (xs @ ys) = length xs + length ys
proof (induct xs)
case Nil
  show "length ([]) @ ys) = length [] + length ys"
    by simp
next
case (Cons x xs)
  have ih: "length (xs@ys) = length xs + length ys" by fact
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
    using ih by simp
qed

We know by definition
- \( (x \# xs) \@ ys = x \# xs \@ ys \)
- \( \text{length} \ (x \# xs) = 1 + \text{length} \ xs \)

(remember for all \( x \) and \( xs \))
Proof by Induction

lemma length_append_test:
  shows "length (xs @ ys) = length xs + length ys"
proof (induct xs rule: list.induct)
  case Nil
  show "length ([] @ ys) = length [] + length ys"
    by simp
next
  case (Cons x xs)
  have ih: "length (xs@ys) = length xs + length ys" by fact
  show "length ((x#xs)@ys) = length (x#xs) + length ys"
    using ih by simp
qed
Proof by Induction

lemma length_append_test:
  shows "length (xs @ ys) = length xs + length ys"
by (induct xs) (simp_all)

- Isabelle can figure out in most cases which induction to use.
- simp_all works on all subgoals (simp only on the first).
Another Proof

\textbf{lemma} \texttt{length\_rev\_test}:
  \texttt{shows "length (rev xs) = length xs"}
\texttt{sorry}

\begin{itemize}
  \item Let’s do it first by hand.
\end{itemize}
Another Proof

lemma length_rev_test:
  shows "length (rev xs) = length xs"
proof (induct xs)
case Nil
  show "length (rev []) = length []" by simp
next
case (Cons x xs)
  have ih: "length (rev xs) = length xs" by fact
  have aux: "length ((rev xs)@[x]) =
    length (rev xs) + length [x]"
    by (simp add: length_append)
  show "length (rev (x#xs)) = length (x#xs)"
    using ih aux by simp
qed

In this proof we need an auxiliary fact.
A Few More Proofs

lemma append_assoc:
  shows "(xs @ ys) @ zs = xs @ (ys @ zs)"
by (induct xs) (simp_all)

lemma append_Nil2:
  shows "xs @ [] = xs"
by (induct xs) (simp_all)

lemma rev_append:
  shows "rev (xs @ ys) = (rev ys) @ (rev xs)"
by (induct xs) (simp_all add: append_Nil2 append_assoc)

lemma rev_rev_ident:
  shows "rev (rev xs) = xs"
by (induct xs) (simp_all add: rev_append)

---
Be careful: the simplifier might also loop.
So far we only had functions that are defined for all their arguments. In many circumstances it is helpful to indicate an error for certain inputs. One way to achieve this is by using `options`.

```lean
datatype 'a option =
  None
  | Some ""a"
```

What does the induction principle for option look like? (Try `thm`!)
fun lookup :: "'a ⇒ ('a × 'b) list ⇒ 'b option"
where
  "lookup x [] = None"
| "lookup x ((y,z)#xs) = (if x=y then Some z else lookup x xs)"

Note the type and see how it reflects the intended behaviour.

What do you say to this lemma?

lemma lookup_test:
  assumes "(x,y) ∈ set xs"
  shows "lookup x xs = Some y"
Point to Take Home

We did **structural inductions** over lists.

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

Isabelle will automatically provide us with

\[
\begin{align*}
P \text{ Nil} & \quad \forall x \text{ xs. } P \text{ xs} \implies P (\text{Cons } x \text{ xs}) \\
\forall x \text{ xs. } P \text{ xs}
\end{align*}
\]
We did structural inductions over lists.

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

Isabelle will automatically provide us with

\[
P \text{Nil} \quad \land \quad \forall x \, \text{xs}. \quad P \, \text{xs} \implies P \, (\text{Cons} \, x \, \text{xs})
\]

Try \texttt{thm list.induct[no_vars]}