Semantics

of Programming Languages (4)

Christian Urban

http://www4.in.tum.de/~urbanc/Teaching/semantics08.html
Automated proving is not just a slightly more fussy version of paper proving... It’s a strange new skill, much harder to learn than a new programming language or application, or even many bits of mathematics. I’m resistant to investing significant effort in tools (I don’t write clever TeX or Emacs macros), but the payoff really came the second time I used Coq: I was able to prove some elementary but delicate results for a different paper in just a day or so. Coq is worth the bother and it, or something like it, is the future, if only we could make the initial learning experience a few thousand times less painful.
lemma test:
  fixes k::"nat"
  shows "2 * k = k + k"
  by simp

thm test
  \[\Rightarrow 2 * ?k = ?k + ?k\]

thm test[no_vars]
  \[\Rightarrow 2 * k = k + k\]

thm test[symmetric,no_vars]
  \[\Rightarrow k + k = 2 * k\]
Chaining of Facts

A version not using labels, but using then:

lemma odd_and_even:
  assumes a: "even n ∧ odd n"
  shows "False"
using a
proof (induct n)
  case 0
  have "even 0 ∧ odd 0" by fact
  then have "odd 0" by simp
  then show "False"
    by (cases rule: odd.cases) (auto)
next
  case (Suc n)…
Chaining of Facts

A version using labels and using (post-fix):

```
lemma odd_and_even:
  assumes a: "even n ∧ odd n"
  shows "False"
using a
proof (induct n)
  case 0
    have a1: "even 0 ∧ odd 0" by fact
    have a2: "odd 0" using a1 by simp
    show "False" using a2
      by (cases rule: odd.cases) (auto)
  next
  case (Suc n)...
```
A version using labels and `from` (pre-fix):

```plaintext
lemma odd_and_even:
  assumes a: "even n ∧ odd n"
  shows "False"
using a
proof (induct n)
  case 0
    have a1: "even 0 ∧ odd 0" by fact
    from a1 have a2: "odd 0" by simp
    from a2 show "False"
      by (cases rule: odd.cases) (auto)
next
  case (Suc n)...```
A Sequence of Facts

have l1: “…”
have l2: “…”

…

have ln: “…”
from l1 l2 … ln have “…”

works also for show

have “…”
moreover have “…”

…

moreover have “…”
ultimately have “…”

Semantics, 29. April 2008 - p.7 (1/1)
lemma odd_and_even:
   assumes a: "even n ∧ odd n"
   shows "False"
using a proof (induct n)
...
next
  case (Suc n)
  have ih: "even n ∧ odd n ⟷ False" by fact
  have as: "even (Suc n) ∧ odd (Suc n)" by fact
  from as have "odd (Suc n)" by simp
  then have "even (Suc (Suc n))" ...
  then have "even ((Suc (Suc n)) - 2)" ...
  then have A: "even n" by simp
  from as have "even (Suc n)" by simp
  then have "odd (Suc (Suc n))" ...
  moreover have "(Suc (Suc n)) > 1" by simp
  ultimately have "odd ((Suc (Suc n)) - 2)" ...
  then have B: "odd n" by simp
  show "False" using ih A B by simp
qed
lemma odd_or_even:
  shows "even n \lor odd n"
proof (induct n)
  
next
  case (Suc n)
  have ih: "even n \lor odd n" by fact
  moreover
  { assume a1: "even n"
    have a2: "odd (Suc n)" using a1 by (simp add: odd_suc)
    have "even (Suc n) \lor odd (Suc n)" using a2 by simp
  }
  moreover
  { assume b1: "odd n"
    have b2: "even (Suc n)" using b1 by (simp add: even_suc)
    have "even (Suc n) \lor odd (Suc n)" using b2 by simp
  }
  ultimately show "even (Suc n) \lor odd (Suc n)" by blast
qed
lemma odd_or_even:
  shows "even n ∨ odd n"
proof (induct n)
  ... 
next 
  case (Suc n)
  have ih: "even n ∨ odd n" by fact
  moreover
  { assume "even n"
    then have "odd (Suc n)" by (simp add: odd_suc)
    then have "even (Suc n) ∨ odd (Suc n)" by simp
  }
  moreover
  { assume "odd n"
    then have "even (Suc n)" by (simp add: even_suc)
    then have "even (Suc n) ∨ odd (Suc n)" by simp
  }
  ultimately show "even (Suc n) ∨ odd (Suc n)" by blast
qed
Case Distinctions

have “\( P_1 \vee P_2 \vee P_3 \)” …
moreover
{ assume “\( P_1 \)”
  …
  have “something”… }
moreover
{ assume “\( P_2 \)”
  …
  have “something”… }
moreover
{ assume “\( P_3 \)”
  …
  have “something”… }
ultimately have “something” by blast
You can also nest proofs (instead of by)

\begin{verbatim}
lemma "something"
proof (…)
  …
  have "something else"
  proof …
  …
  qed
  …
  show "something"
  proof …
  …
  qed
qed
\end{verbatim}
A Rough Grammar

proof ::= proof [method] statements* qed
  | by method
  | by method method

method ::= (simp...) | (auto...)
  | (induct...) |

statement ::= assume proposition
  | [from name+] have/show
  proposition [using name+] proof
  | next

proposition ::= [name:] formula
Setting up Inductions

Rule inductions or inductions over an inductive definition are of the form

```
lemma
  assumes a: "rel args"
  shows "something"
```

(rel is the inductively defined predicate)

The proof method is then

```
using a proof (induct)
```

or (induct args) or (induct rule: ...)

Semantics, 29. April 2008 - p.14 (1/1)
Setting up Inductions

Simple structural inductions are of the form

\[
\text{lemma}
\text{shows \textquotedblleft something (arg)\textquotedblright}
\]

The proof method is then

\[
\text{proof (induct arg)}
\]

or (induct arg rule: \ldots)
Setting up Inductions

Structural inductions can be also of the form

**lemma**

assumes a: “smth harmless”

shows “something (arg)”

Then the proof method will still work as expected

using a proof (induct arg)
However, if you want to do a structural induction in the case

\textbf{lemma}

\texttt{assumes a: \textasciitilde{rel (arg)}}

\texttt{shows \textasciitilde{something (arg)}}

\textbf{you must give explicitly the induction rule}

\texttt{using a proof (induct arg rule: \ldots )}
If you have the inductive definition

```plaintext
inductive
  even :: "nat ⇝ bool"
where
  even_zero: "even 0"
  | even_step: "even n ⇝ even (Suc (Suc n))"
```

Then the case names in the rule induction are the names of the rules. So you better give always rules an explicit name (otherwise cases will get numbers: 0 1).
Any questions about the HW?

(apart from “What to use: auto, simp, blast…” ;o)
thm conjI

\[[P; \ Q] \implies P \land Q\]

[...] represents assumptions

"if I know P and Q, I can deduce P \land Q"

stands for the rule

\[
\begin{array}{c}
P \quad Q \\
\hline
P \land Q
\end{array}
\]
Isar Basics (2)

lemma
  assumes a: "P" "Q"
  shows "P ∧ Q"
using a by (auto)

lemma
  assumes a: "P" "Q"
  shows "P ∧ Q"
using a by (simp)

lemma
  assumes a: "P" "Q"
  shows "P ∧ Q"
using a by (blast)
lemma
assumes a: "P" "Q"
shows "P ∧ Q"
using a
proof -
  have "P" by fact
moreover
  have "Q" by fact
ultimately show "P ∧ Q" by (rule conjI)
qed

proof - is "do nothing"

[P; Q] ⟷ P ∧ Q
lemma
  assumes a: "P"
  shows "P ∨ Q"
proof -
  have "P" by fact
  then show "P ∨ Q" by (rule disjI1)
qed

thm disjI1 gives P ⟷ P ∨ Q
thm disjI2 gives Q ⟷ P ∨ Q
lemma
  assumes a: "A" "B" "C"
  shows "(A ∧ B) ∧ C"
proof -
  have a: "C" by fact
  have "A" by fact
  moreover
  have "B" by fact
  ultimately
  have "A ∧ B" by (rule conjI)
  then show "(A ∧ B) ∧ C" using a by (rule conjI)
qed
lemma
assumes a: “A ∧ B”
shows “B ∧ A”
using a
proof -
  have a: “A ∧ B” by fact
  have “B” using a by (rule conjunct2)
moreover
  have “A” using a by (rule conjunct1)
ultimately
  show “B ∧ A” by (rule conjI)

(order of chained facts is important)

qed

thm conjunct1 gives P ∧ Q ⟹ P
thm conjunct2 gives P ∧ Q ⟹ Q
in case $A \Rightarrow A$ use: by (assumption)
(i.e. an assumption exactly matches with the goal)

lemma
shows "$A \Rightarrow A$"
proof -
\[
\begin{array}{l}
\{ \text{assume } "A" \\
\quad \text{then have } "A" \text{ by assumption} \\
\} \\
\text{then show } "A \Rightarrow A" \text{ by (rule impI)}
\end{array}
\]
qed

\[
\begin{array}{c}
\begin{array}{c}
A \\
\hline
A
\end{array}
\end{array}
\]

assumption
Notice the subtle difference:

**Lemma**

shows \( A \implies A \)
by (rule `impI`) (assumption)

**Lemma**

shows \( A \iff A \)
by (assumption)

**Lemma**

assumes \( a: A \)
shows \( A \)
using \( a \) by (assumption)

\( \iff \) is the meta-implication (essentially) separating assumptions and the goal; \( \implies \) is the usual implication.
lemma
  assumes a: "P ∨ Q"
  and b₁: "P ⊢ C" and b₂: "Q ⊢ C"
  shows "C"
using a
proof -
  have "P ∨ Q" by fact
  moreover
  { assume "P"
    then have "C" by (rule b₁) }
  moreover
  { assume "Q"
    then have "C" by (rule b₂) }
  ultimately
  show "C" by (rule disjE)
qed

Note that (rule...) can take anything of the form _ → _.
lemma
assumes a: “P ∨ Q”
and b₁: “P \implies C” and b₂: “Q \implies C”
shows “C”
using a b₁ b₂ by (rule disjE)

\[
\text{thm disjE gives } [P \lor Q; P \implies R; Q \implies R] \implies R
\]

lemma
assumes a: “P ∨ Q”
and b₁: “P \implies C” and b₂: “Q \implies C”
shows “C”
using b₁ b₂ a oops

for (rule…) the order of the facts is important
**Point to Take Home**

- **blast** is just applying rules in a very clever way; it uses a set of predefined lemmas (e.g. conjI, allI, disjE...).

- About **simp** and **auto** it is more difficult to say what they precisely doing (optimised over many years).

- **simp** is not just using equations, but also analyses conditional equations.

- I can give you rules of thumbs, but you have to get a feeling for them.

- Tomorrow, I will give you hints about “debugging” for when things go wrong.