Interactive Formal Verification
4: Advanced Recursion, Induction and Simplification

Tjark Weber
(Slides: Lawrence C Paulson)
Computer Laboratory
University of Cambridge
A Failing Proof by Induction
A Failing Proof by Induction

fun itlen :: "a list => nat => nat" where
  "itlen Nil n = n"
| "itlen (Cons x xs) n = itlen xs (Suc n)"

lemma "itlen xs n = size xs + n"
apply (induct xs)
apply auto
oops

proof (prove): step 2

goal (1 subgoal):
1. \( \forall xs. \) itlen \( xs \) \( n = \) size \( xs \) + \( n \) \( \Rightarrow \) itlen \( xs \) (Suc \( n \)) = Suc (size \( xs \) + \( n \))

*goals*
A Failing Proof by Induction

length of a list (tail-recursive)
equivalent to the built-in length function?
A Failing Proof by Induction

length of a list (tail-recursive)
equivalent to the built-in length function?

Mismatch between induction hypothesis and conclusion!
A Failing Proof by Induction

May as well give up!

Mismatch between induction hypothesis and conclusion!

length of a list (tail-recursive)
equivalent to the built-in length function?
Generalising the Induction

Insert a universal quantifier

Induction hypothesis holds for all n
Generalising: Another Way
Generalising: Another Way

Designate a variable as “arbitrary”

fun itlen :: "a list ⇒ nat ⇒ nat" where
"itlen Nil n = n"
| "itlen (Cons x xs) n = itlen xs (Suc n)"

lemma "itlen xs n = size xs + n"
apply (induct xs arbitrary: n)
apply auto
done

proof (prove): step 1

goal (2 subgoals):
1. ∀n. itlen Nil n = size Nil + n
2. ∀a xs n.
   (∀n. itlen xs n = size xs + n) ⇒
   itlen (Cons a xs) n = size (Cons a xs) + n
Generalising: Another Way

Designate a variable as “arbitrary”

Induction hypothesis still holds for all $n$!
Unusual Recursions

subsection { * Ackermann's Function * }

fun ack :: "nat => nat => nat" where
  "ack 0 n = Suc n"
  "ack (Suc m) 0 = ack m 1"
  "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

lemma less_ack2 [iff]: "j < ack i j"
apply (induct i j rule: ack.induct)
apply auto

proof (prove): step 1

goal (3 subgoals):
1. \n. n < ack 0 n
2. \m. 1 < ack m 1 ==> 0 < ack (Suc m) 0
3. \m n. [(n < ack (Suc m) n; ack (Suc m) n < ack m (ack (Suc m) n)]
   ==> Suc n < ack (Suc m) (Suc n)
Unusual Recursions

Two variables in the recursion!
Unusual Recursions

Two variables in the induction!

Two variables in the recursion!
Unusual Recursions

Two variables in the induction!

Two variables in the recursion!

A special induction rule!
Unusual Recursions

Two variables in the induction!

Two variables in the recursion!

A special induction rule!

The subgoals follow the recursion!
Recursion: Key Points
Recursion: Key Points

- Recursion in one variable, following the structure of a datatype declaration, is called *primitive*. 
Recursion: Key Points

- Recursion in one variable, following the structure of a datatype declaration, is called *primitive*.
- Recursion in multiple variables, terminating by size considerations, can be handled using `fun`.
  - `fun` produces a special induction rule.
  - `fun` can handle *nested recursion*.
  - `fun` also handles *pattern matching*, which it completes.
Special Induction Rules
Special Induction Rules

• They follow the function’s recursion exactly.
Special Induction Rules

• They follow the function’s recursion exactly.

• For Ackermann, they reduce $P x y$ to
  
  • $P 0 n$, for arbitrary $n$
  
  • $P (Suc m) 0$ assuming $P m 1$, for arbitrary $m$
  
  • $P (Suc m) (Suc n)$ assuming $P (Suc m) n$ and $P m (ack (Suc m) n)$, for arbitrary $m$ and $n$
Special Induction Rules

• They follow the function’s recursion exactly.

• For Ackermann, they reduce $P \, x \, y$ to
  • $P \, 0 \, n$, for arbitrary $n$
  • $P \, (Suc \, m) \, 0$ assuming $P \, m \, 1$, for arbitrary $m$
  • $P \, (Suc \, m) \, (Suc \, n)$ assuming $P \, (Suc \, m) \, n$ and $P \, m \, (ack \, (Suc \, m) \, n)$, for arbitrary $m$ and $n$

• **Usually** they do what you want. Trial and error is tempting, but ultimately you will need to think!
Another Unusual Recursion

```haskell
fun merge :: "'a list ⇒ 'a list ⇒ 'a list"
where
  "merge (x#xs) (y#ys) = (if x ≤ y then x # merge xs (y#ys) else y # merge (x#xs) ys)"
| "merge xs [] = xs"
| "merge [] ys = ys"

lemma set_merge[simp]: "set (merge xs ys) = set xs ∪ set ys"
apply (induct xs ys rule: merge.induct)
apply auto
done
```

proof (prove): step 1

goal (3 subgoals):
1. ∀x xs y ys. [x ≤ y ⇒ set (merge xs (y # ys)) = set xs ∪ set (y # ys);
   ¬ x ≤ y ⇒ set (merge (x # xs) (y # ys)) = set (x # xs) ∪ set ys]
   ⇒ set (merge (x # xs) (y # ys)) = set (x # xs) ∪ set (y # ys)
2. ∀xs. set (merge xs []) = set xs ∪ set []
3. ∀v va. set (merge [] (v # va)) = set [] ∪ set (v # va)

---

*goals*
Top L1 (Isar Proofstate Utoks Abbrev;)-----------------------------
Wrote /Users/lp15/Dropbox/ACS/4 - Advanced Recursion/MergeSort.thy
Another Unusual Recursion

recursive calls are guarded by conditions

fun merge :: "'a list ⇒ 'a list ⇒ 'a list"
where
  "merge (x#xs) (y#ys) = (if x ≤ y then x # merge xs (y#ys) else y # merge (x#xs) ys)"
| "merge xs [] = xs"
| "merge [] ys = ys"

lemma set_merge[simp]: "set (merge xs ys) = set xs ∪ set ys"
apply (induct xs ys rule: merge.induct)
apply auto
done

proof (prove): step 1

goal (3 subgoals):
1. ∀x xs y ys. 
   [x ≤ y ⇒ set (merge xs (y # ys)) = set xs ∪ set (y # ys);
    ¬x ≤ y ⇒ set (merge (x # xs) ys) = set (x # xs) ∪ set ys] 
    ⇒ set (merge (x # xs) (y # ys)) = set (x # xs) ∪ set (y # ys)
2. ∀xs. set (merge xs []) = set xs ∪ set []
3. ∀v va. set (merge [] (v # va)) = set [] ∪ set (v # va)
Another Unusual Recursion

Recursive calls are guarded by conditions!

2 induction hypotheses, guarded by conditions!
Proof Outline

\[ \text{set } (\text{merge } (x#xs) (y#ys)) = \text{set } (x \# xs) \cup \text{set } (y \# ys) \]

\[ \text{set } (\text{if } x \leq y \text{ then } x \# \text{merge } xs (y#ys) \]
\[ \quad \text{else } y \# \text{merge } (x#xs) ys) = \ldots \]
\[ = \]
\[ (x \leq y \rightarrow \text{set}(x \# \text{merge } xs (y#ys)) = \ldots) \quad \& \]
\[ (\neg x \leq y \rightarrow \text{set}(y \# \text{merge } (x#xs) ys) = \ldots) \]
\[ = \]
\[ (x \leq y \rightarrow \{x\} \cup \text{set}(\text{merge } xs (y#ys)) = \ldots) \quad \& \]
\[ (\neg x \leq y \rightarrow \{y\} \cup \text{set}(\text{merge } (x#xs) ys) = \ldots) \]
\[ = \]
\[ (x \leq y \rightarrow \{x\} \cup \text{set } xs \cup \text{set } (y \# ys) = \ldots) \quad \& \]
\[ (\neg x \leq y \rightarrow \{y\} \cup \text{set } (x \# xs) \cup \text{set } ys = \ldots) \]
Proof Outline

\[
\text{set } \text{merge} (x\#xs) (y\#ys) = \text{set } (x \# xs) \cup \text{set } (y \# ys)
\]

\[
\text{set } \text{if } x \leq y \text{ then } x \# \text{merge } xs (y\#ys)
\]
\[
\text{else } y \# \text{merge } (x\#xs) \text{ ys}) = \ldots
\]
\[
= (x \leq y \rightarrow \text{set}(x \# \text{merge } xs (y\#ys)) = \ldots) \&
\]
\[
(\neg x \leq y \rightarrow \text{set}(y \# \text{merge } (x\#xs) \text{ ys}) = \ldots)
\]
\[
= (x \leq y \rightarrow \{x\} \cup \text{set}(\text{merge } xs (y\#ys)) = \ldots) \&
\]
\[
(\neg x \leq y \rightarrow \{y\} \cup \text{set}(\text{merge } (x\#xs) \text{ ys}) = \ldots)
\]
\[
= (x \leq y \rightarrow \{x\} \cup \text{set } xs \cup \text{set } (y \# ys) = \ldots) \&
\]
\[
(\neg x \leq y \rightarrow \{y\} \cup \text{set } (x \# xs) \cup \text{set } ys = \ldots)
\]
Proof Outline

\[ \text{set } (\text{merge } (x \# xs) (y \# ys)) = \text{set } (x \# xs) \cup \text{set } (y \# ys) \]

\[ \text{set } (\begin{array}{l}
\text{if } x \leq y \text{ then } x \# \text{merge } xs (y \# ys) \\
\text{else } y \# \text{merge } (x \# xs) ys
\end{array}) = \ldots 
\]

\[ = (x \leq y \rightarrow \text{set}(x \# \text{merge } xs (y \# ys)) = \ldots) \& \\
(\neg x \leq y \rightarrow \text{set}(y \# \text{merge } (x \# xs) ys) = \ldots) 
\]

\[ = (x \leq y \rightarrow \{x\} \cup \text{set}(\text{merge } xs (y \# ys)) = \ldots) \& \\
(\neg x \leq y \rightarrow \{y\} \cup \text{set}(\text{merge } (x \# xs) ys) = \ldots) 
\]

\[ = (x \leq y \rightarrow \{x\} \cup \text{set } xs \cup \text{set } (y \# ys) = \ldots) \& \\
(\neg x \leq y \rightarrow \{y\} \cup \text{set } (x \# xs) \cup \text{set } ys = \ldots) \]
Proof Outline

\[\text{set } \left( \text{merge } (x\#xs) \ (y\#ys) \right) = \text{set } (x \ # \ xs) \cup \text{set } (y \ # \ ys)\]

\[\text{set } \left( \begin{array}{l}
\text{if } x \leq y \text{ then } x \ # \ \text{merge } xs \ (y\#ys) \\
\text{else } y \ # \ \text{merge } (x\#xs) \ ys
\end{array} \right) = \ldots\]

\[=\]

\[\begin{array}{c}
(x \leq y \rightarrow \text{set}(x \ # \ \text{merge } xs \ (y\#ys)) = \ldots) \& \\
(\neg x \leq y \rightarrow \text{set}(y \ # \ \text{merge } (x\#xs) \ ys) = \ldots)
\end{array}\]

\[=\]

\[\begin{array}{c}
(x \leq y \rightarrow \{x\} \cup \text{set}(\text{merge } xs \ (y\#ys)) = \ldots) \& \\
(\neg x \leq y \rightarrow \{y\} \cup \text{set}(\text{merge } (x\#xs) \ ys) = \ldots)
\end{array}\]

\[=\]

\[\begin{array}{c}
(x \leq y \rightarrow \{x\} \cup \text{set } xs \cup \text{set } (y \ # \ ys) = \ldots) \& \\
(\neg x \leq y \rightarrow \{y\} \cup \text{set } (x \ # \ xs) \cup \text{set } ys = \ldots)
\end{array}\]
Proof Outline

\[
\text{set (merge (x#xs) (y#ys)) = set (x # xs) } \cup \text{ set (y # ys)}
\]

\[
\text{set (if x \leq y then x # merge xs (y#ys) else y # merge (x#xs) ys)} = ...
\]

\[
= (x \leq y \rightarrow \text{set(x # merge xs (y#ys)) = ...)} \&
(\neg x \leq y \rightarrow \text{set(y # merge (x#xs) ys) = ...})
\]

\[
= (x \leq y \rightarrow \{x\} \cup \text{set(merge xs (y#ys)) = ...)} \&
(\neg x \leq y \rightarrow \{y\} \cup \text{set(merge (x#xs) ys) = ...})
\]

\[
= (x \leq y \rightarrow \{x\} \cup \text{set xs } \cup \text{set (y # ys) = ...)} \&
(\neg x \leq y \rightarrow \{y\} \cup \text{set (x # xs) } \cup \text{set ys = ...})
\]
The Case Expression
The Case Expression

- Similar to that found in the functional language ML.
The Case Expression

- Similar to that found in the functional language ML.
- Automatically generated for every datatype.
The Case Expression

- Similar to that found in the functional language ML.
- Automatically generated for every datatype.
- The simplifier can (upon request!) perform case-splits analogous to those for “if”.
The Case Expression

• Similar to that found in the functional language ML.
• Automatically generated for every datatype.
• The simplifier can (upon request!) perform case-splits analogous to those for “if”.
• Case splits in *assumptions* (not the conclusion) never happen unless requested.
Case-Splits for Lists
fun ordered :: "'a list => bool"
where
  "ordered [] = True"
| "ordered [x] = True"
| "ordered (x#y#xs) = (x≤y & ordered (y#xs))"
fun ordered :: "'a list => bool"
where
  "ordered [] = True"
| "ordered (x#l) =
    (case l of [ ] => True
     | Cons y xs => (x≤y & ordered y#xs)))"
Case-Splitting in Action

Help! Look at all the case-splits!
Case-Splitting in Action

Automatic case splitting to the rescue!

Help! Look at all the case-splits!
Completing the Proof

```plaintext
lemma ordered_merge [simp]: "ordered (merge xs ys) = (ordered xs & ordered ys)"
apply (induct xs ys rule: merge.induct)
apply simp_all
apply (auto split: list.split
simp del: ordered.simps(2))
```

proof (prove): step 3

goal:
No subgoals!
Completing the Proof

All solved, in two seconds.
Completing the Proof

All solved, in two seconds.

But what is this? Risk of looping!
Case Splitting for Lists

Simplification will replace

\[ P (\text{case } xs \text{ of } [] \Rightarrow a \mid \text{Cons } h t l \Rightarrow b \ h \ t l) \]

by

\[ (xs = [ ] \rightarrow P a) \land (\forall h t l. xs = h \ # t l \rightarrow P (b \ h \ t l)) \]
Case Splitting for Lists

Simplification will replace

\[ P \left( \text{case } xs \text{ of } [] => a \mid \text{Cons } h \ tl \Rightarrow b \ h \ tl \right) \]

by

\[(xs = [ ] \rightarrow P a) \land (\forall h \ tl. \ xs = h \# \ tl \rightarrow P (b \ h \ tl))\]

- It creates a case for each datatype constructor.
Case Splitting for Lists

Simplification will replace

\[ P \left( \text{case } xs \text{ of } [ ] \Rightarrow a \mid \text{Cons } h t l \Rightarrow b h t l \right) \]

by

\[ (xs = [ ] \rightarrow P a) \land (\forall h t l. xs = h \# t l \rightarrow P (b h t l)) \]

- It creates a case for each datatype constructor.
- Here it causes looping if combined with the second rewrite rule for ordered.
Summary
Summary

• Many forms of recursion are available.
Summary

- Many forms of recursion are available.
- The supplied induction rule often leads to simple proofs.
Summary

• Many forms of recursion are available.

• The supplied induction rule often leads to simple proofs.

• The “case” operator can often be dealt with using automatic case splitting...
Summary

- Many forms of recursion are available.
- The supplied induction rule often leads to simple proofs.
- The “case” operator can often be dealt with using automatic case splitting...
- but complex simplifications can run forever!
A Helpful Tip