CS477 Formal Software Dev Methods

Elsa L Gunter
2112 SC, UIUC
egunter@illinois.edu
http://courses.engr.illinois.edu/cs477

Slides based in part on previous lectures
by Mahesh Vishwanathan, and by Gul Agha

February 13, 2020

Proof by Assumption

$$\frac{\mathtt{A_1 \ldots A_i \ldots A_n}}{\mathtt{A_i}}$$

- Proof method: `assumption`
- Use:

  `apply assumption`

- Proves:

  $[\![\mathtt{A_1; \ldots ; A_n}]\!] \Longrightarrow \mathtt{A}$

  by unifying A with one of the A_i

Rule Application: The Rough Idea

- Applying rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ to subgoal C:
  - Unify A and C
  - Replace C with n new subgoals: $A'_1 \ \ldots \ A'_n$
- Backwards reduction, like in Prolog
- Example:
  Rule: $[\![\mathtt{?P; ?Q}]\!] \Longrightarrow \mathtt{?P \wedge ?Q}$
  Subgoal: 1. $\mathtt{A \wedge B}$
- Resulting Subgoals :
  1. A
  2. B

Rule Application: More Complete Idea

Applying rule $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$ to subgoal C:

- Unify A and C with (meta)-substitution σ
- Specialize goal to $\sigma(C)$
- Replace C with n new subgoals: $\sigma(A_1) \ \ldots \ \sigma(A_n)$

Note: schematic variables in C treated as existential variables
Does there exist value for $?X$ in C that makes C true?
(Still not the whole story)

`rule` Application

Rule: $[\![A_1; \ldots ; A_n]\!] \Longrightarrow A$

Subgoal: 1. $[\![B_1; \ldots ; B_m]\!] \Longrightarrow C$

Substitution: $\sigma(A) \equiv \sigma(C)$

New subgoals: 1. $[\![\sigma(B_1); \ldots ; \sigma(B_m)]\!] \Longrightarrow \sigma(A_1)$
$\vdots$
n. $[\![\sigma(B_1); \ldots ; \sigma(B_m)]\!] \Longrightarrow \sigma(A_n)$

Proves: $[\![\sigma(B_1); \ldots ; \sigma(B_m)]\!] \Longrightarrow \sigma(C)$

Command: `apply (rule` <*rulename*>`)`

Applying Elimination Rules

`apply (erule` <*elim-rule*>`)`

Like `rule` but also

- Unifies first premise of rule with an assumption
- Eliminates that assumption instead of conclusion

Example

Rule: $[\![?P \wedge ?Q; [\![?P; ?Q]\!] \Longrightarrow ?R]\!] \Longrightarrow ?R$

Subgoal: 1. $[\![X; A \wedge B; Y]\!] \Longrightarrow Z$

Unification: $?P \wedge ?Q \equiv A \wedge B$ and $?R \equiv Z$
$\{?P \mapsto A;\ ?Q \mapsto B;\ ?R \mapsto Z\}$

New subgoal: 1. $[\![X; Y]\!] \Longrightarrow [\![A; B]\!] \Longrightarrow Z$

Same as: 1.$[\![X; Y; A; B]\!] \Longrightarrow Z$

Defining Things

Introducing New Types

- typedef: Primitive for type definitions; Only real way of introducing a new type with new properties
  - Must build a model and prove it nonempty
  - Probably won't use in this course
- typedecl: Pure declaration; New type with no properties (except that it is non-empty)
- type_synonym: Abbreviation - used only to make theory files more readable
- datatype: Defines recursive data-types; solutions to free algebra specifications

Datatypes: An Example

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

- Type constructors: list of one argument
- Term constructors: `Nil :: 'a list`
  `Cons :: 'a ⇒ 'a list ⇒ 'a list`
- Distinctness: `Nil` $\neq$ `Cons x xs`
- Injectivity:
  `(Cons x xs = Cons y ys) = (x = y ∧ xs = ys)`

Structural Induction on Lists

- To show P holds of every list
  - show P Nil, and
  - for arbitrary a and list, show P list implies P (Cons a list)

$$\frac{\texttt{P Nil} \quad \begin{matrix}\texttt{P list} \\ \vdots \\ \texttt{P (Cons a list)}\end{matrix}}{\texttt{P xs}}$$

In Isabelle:

$[\![?P\ [\,];\ \bigwedge a\ list.\ ?P\ list \Longrightarrow ?P\ (a \# list)]\!] \Longrightarrow ?P\ ?list$

datatype: The General Case

$$\texttt{datatype}\ (\alpha_1, \ldots, \alpha_m)\tau \ = \ C_1\ \tau_{1,1} \ldots \tau_{1,n_1} \ \mid \ \ldots \ \mid \ C_k\ \tau_{k,1} \ldots \tau_{k,n_k}$$

- Term Constructors:
  $C_i :: \tau_{i,1} \Rightarrow \ldots \Rightarrow \tau_{i,n_i} \Rightarrow (\alpha_1, \ldots, \alpha_m)\tau$
- Distinctness: $C_i\ x_i \ldots x_{i,n_i} \neq C_j\ y_j \ldots y_{j,n_j}$ if $i \neq j$
- Injectivity: $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) =$
  $(x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

Distinctness and Injectivity are applied by simp
Induction must be applied explicitly

Proof Method

- **Syntax:** (induct_tac x)
  x must be a free variable in the first subgoal
  The type of x must be a datatype
- **Effect:** Generates 1 new subgoal per constructor
- Type of x determines which induction principle to use

case

Every datatype introduces a case construct, e.g.

(case xs of [ ] ⇒...| y#ys ⇒ ...y ...ys ...)

In general: case *Arbitrarily nested pattern* ⇒ *Expression using pattern variables* | ...
Patterns may be non-exhaustive, or overlapping
Order of clauses matters - early clause takes precedence.

HOL Functions are Total

Why nontermination can be harmful:

- If f x is undefined, is f x = f x?
- Excluded Middle says it must be True or False
- Reflexivity says it's True
- How about f x = 0? f x = 1? f x = y?
- If f x ≠ y then ∀y. f x ≠ y.
- Then f x ≠ f x #

! All functions in HOL must be total !

Function Definition in Isabelle/HOL

- Non-recursive definitions with definition
  No problem
- Well-founded recursion with fun
  Proved automatically, but user must take care that recursive calls are on "obviously" smaller arguments
- Well-founded recursion with function
  User must (help to) prove termination
  (⇝ later)
- Role your own, via definition of the functions graph
  use of choose operator, and other tedious approaches, but can work when built-in methods don't.
- Shouldn't need last two in this class

A Recursive Function: List Append

Declaration:
consts app :: "'a list ⇒ 'a list ⇒ 'a list
and definition by *recursion*:

```
fun
app Nil ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

Uses heuristics to find termination order
Guarantees termination (total function) if it succeeds

Demo: Another Datatype Example