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 9, 2018
Proof by Assumption

\[
\begin{align*}
& A_1 \ldots A_i \ldots A_n \\
\hline
& A_i
\end{align*}
\]

- Proof method: \textit{assumption}
- Use: \textit{apply assumption}
- Proves: \[
[A_1; \ldots; A_n] \rightarrow A
\]
  by unifying \( A \) with one of the \( A_i \)
Applying rule \([A_1; \ldots ; A_n] \Rightarrow A\) 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:
\[
[?P; ?Q] \Rightarrow ?P \land ?Q
\]
Subgoal:
1. \(A \land B\)

Resulting Subgoals:
1. \(A\)
2. \(B\)
Applying rule $[A_1; \ldots; A_n] \rightarrow A$ to subgoal $C$:

- Unify $A$ and $C$ with (meta)-substitution $\sigma$
- 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:** \[ A_1; \ldots; A_n \Rightarrow A \]

**Subgoal:** 1. \[ B_1; \ldots; B_m \Rightarrow C \]

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

**New subgoals:**
1. \[ \sigma(B_1); \ldots; \sigma(B_m) \Rightarrow \sigma(A_1) \]
   
   :  
   
   n. \[ \sigma(B_1); \ldots; \sigma(B_m) \Rightarrow \sigma(A_n) \]

**Proves:** \[ \sigma(B_1); \ldots; \sigma(B_m) \Rightarrow \sigma(C) \]

**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: \[ [P \land Q; P; Q] \rightarrow R \rightarrow R \rightarrow R \]

Subgoal: 1. \[ X; A \land B; Y \rightarrow Z \]

Unification: \[ P \land Q \equiv A \land B \text{ and } R \equiv Z \]
\[ \{P \rightarrow A; Q \rightarrow B; R \rightarrow Z\} \]

New subgoal: 1. \[ X; Y \rightarrow [A; B] \rightarrow Z \]

Same as: 1. \[ X; Y; A; B] \rightarrow Z \]
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

- **typedefcl**: 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

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

- Type constructors: `list` of one argument
- Term constructors: `Nil :: 'a list`
  ```plaintext
  Cons :: 'a ⇒ 'a list ⇒ 'a list
  ```
- Distinctness: `Nil ≠ Cons x xs`
- Injectivity:
  ```plaintext
  (Cons x xs = Cons y ys) = (x = y ∧ xs = ys)
  ```
Structural Induction on Lists

To show $P$ holds of every list

- show $P \text{ Nil}$, and
- for arbitrary $a$ and $\text{list}$, show $P \text{ list}$ implies $P (\text{Cons } a \text{ list})$

In Isabelle:

\[
\begin{align*}
P \text{ list} \\
\vdash \\
P \text{ Nil} & \quad P (\text{Cons } a \text{ list}) \\
\hline
\vdash P \text{ xs}
\end{align*}
\]

\[
[| ?P []; \lambda a \text{ list. } ?P \text{ list } \Longrightarrow ?P (a \# \text{list}) |] \Longrightarrow ?P \ ? \text{list}
\]
datatype: The General Case

datatype $(\alpha_1, \ldots, \alpha_m) \tau \ = \ C_1 \ \tau_{1,1} \ldots \tau_{1,n_1} \\ \ldots \\ \ 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:**
  \[C_i \ x_i \ldots x_{i,n_i} \neq C_j \ y_j \ldots y_{j,n_j} \text{ if } i \neq j\]

- **Injectivity:**
  \[\left(C_i \ x_1 \ldots x_{n_i} = C_i \ y_1 \ldots y_{n_j}\right) = \left(x_1 = y_1 \land \ldots \land x_{n_i} = y_{n_j}\right)\]

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.
Every **datatype** introduces a **case** construct, e.g.

\[
\text{case } xs \text{ of } [ ] \Rightarrow \ldots \mid y#ys \Rightarrow \ldots y \ldots ys \ldots
\]

In general: **case** *Arbitrarily nested pattern* \( \Rightarrow \) *Expression using pattern variables* \mid \ldots

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 \( f \ x \neq y \) then \( \forall y. \ f \ x \neq y. \)
- Then \( f \ x \neq f \ x \neq \)

! All functions in HOL must be total !
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