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

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

- Proof method: assumption
- Use:

apply assumption

Proves:

 $\llbracket \mathtt{A}_1;\ldots;\mathtt{A}_n \rrbracket \Longrightarrow \mathtt{A}$

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

Subgoal: 1. A ∧ B

- Resulting Subgoals :

Rule Application: More Complete Idea

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

- ullet 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

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

1. $||B_1; \ldots; B_m|| \Longrightarrow C$ Subgoal:

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

New subgoals: 1. $\llbracket \sigma(B_1); \ldots; \sigma(B_m) \rrbracket \Longrightarrow \sigma(A_1)$

 $n. \| \sigma(B_1); \ldots; \sigma(B_m) \| \Longrightarrow \sigma(A_n)$

Proves: $\llbracket \sigma(B_1); \ldots; \sigma(B_m) \rrbracket \Longrightarrow \sigma(C)$ 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

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

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

Unification: $?P \land ?Q \equiv A \land B \text{ and } ?R \equiv Z$

 $\{?P \mapsto A; ?Q \mapsto B; ?R \mapsto Z\}$

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

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

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
- datatype: Defines recursive data-types; solutions to free algebra specifications

Datatypes: An Example

- Type constructors: list of one argument
- Term constructors: Nil :: 'a list

Cons :: 'a \Rightarrow 'a list \Rightarrow 'a list

- Distinctness: $Nil \neq Cons \times xs$
- Injectivity:

(Cons x xs = Cons y ys) = $(x = y \land xs = ys)$

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)

P list P Nil P (Cons a list)

P xs In Isabelle:

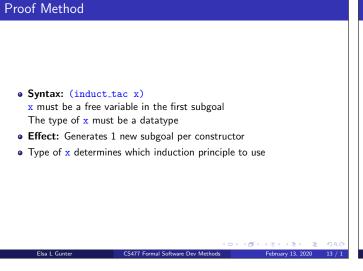
[|?P[]; Aa list. ?P list \Longrightarrow ?P (a#list)|] \Longrightarrow ?P ?list

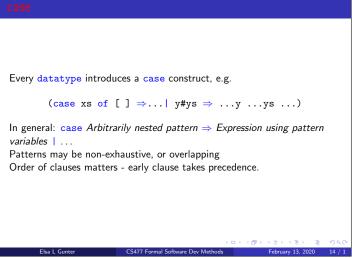
datatype: The General Case

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

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

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





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 $f x \neq f x \#$

! 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 \Rightarrow 'a list \Rightarrow 'a list

and definition by recursion: 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