Interactive Formal Verification 2: Isabelle Theories

Tjark Weber
(Slides: Lawrence C Paulson)
Computer Laboratory
University of Cambridge
theory BT imports Main begin

datatype 'a bt =
  Lf
| Br 'a ''a bt ''a bt

fun reflect :: ''a bt => ''a bt where
  "reflect Lf = Lf"
| "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
    apply auto
  done
end
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a "'a bt" "'a bt"

fun reflect :: "'a bt => 'a bt" where
    "reflect Lf = Lf"
  | "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
    apply (induct t)
    apply auto
    done

end
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a "'a bt" "'a bt"

fun reflect :: 'a bt => 'a bt where
    "reflect Lf = Lf"
  | "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
    apply auto
  done

end
theory BT imports Main begin

datatype 'a bt =
    Lf
  | Br 'a "'a bt" "'a bt"

fun reflect :: "'a bt => 'a bt" where
    "reflect Lf = Lf"
  | "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
  apply (induct t)
  apply auto
  done

end
theory BT imports Main begin

datatype 'a bt = 
    Lf 
  | Br 'a "'a bt" "'a bt"

fun reflect :: "'a bt => 'a bt" where 
    "reflect Lf = Lf"
  | "reflect (Br a t1 t2) = Br a (reflect t2) (reflect t1)"

lemma reflect_reflect_ident: "reflect (reflect t) = t"
    apply (induct t) 
    apply auto 
    done 

end
Notes on Theory Structure
Notes on Theory Structure

• A theory can *import* any existing theories.
Notes on Theory Structure

- A theory can *import* any existing theories.
- Types, constants, etc., must be *declared before use.*
Notes on Theory Structure

• A theory can *import* any existing theories.

• Types, constants, etc., must be *declared before use*.

• The various declarations and proofs may otherwise appear in any order.
Notes on Theory Structure

• A theory can *import* any existing theories.
• Types, constants, etc., must be *declared before use*.
• The various declarations and proofs may otherwise appear in any order.
• Many declarations can be confined to *local scopes*. 
Notes on Theory Structure

• A theory can *import* any existing theories.
• Types, constants, etc., must be *declared before use*.
• The various declarations and proofs may otherwise appear in any order.
• Many declarations can be confined to *local scopes*.
• A finished theory can be imported by others.
Some Fancy Type Declarations

typedec1 loc  -- "an unspecified type of locations"

type_synonym val   = nat -- "values"
type_synonym state = "loc => val"
type_synonym aexp  = "state => val"
type_synonym bexp  = "state => bool"  -- "functions on states"

datatype
    com = SKIP
        | Assign loc aexp         ("_ := _ " 60)
        | Semi   com com          ("_; _"  [60, 60] 10)
        | Cond   bexp com com     ("IF _ THEN _ ELSE _"  60)
        | While  bexp com         ("WHILE _ DO _"  60)
Some Fancy Type Declarations

typedecl loc  -- "an unspecified type of locations"

type_synonym val   = nat -- "values"
type_synonym state = "loc => val"
type_synonym aexp  = "state => val"
type_synonym bexp  = "state => bool"  -- "functions on states"

datatype
com = SKIP
  | Assign loc aexp       ("_ :== _ " 60)
  | Semi   com com        ("_; _" [60, 60] 10)
  | Cond   bexp com com   ("IF _ THEN _ ELSE _" 60)
  | While  bexp com       ("WHILE _ DO _" 60)
Some Fancy Type Declarations

typedecl loc  -- "an unspecified type of locations"

type_synonym val = nat -- "values"
type_synonym state = "loc => val"
type_synonym aexp = "state => val"
type_synonym bexp = "state => bool"  -- "functions on states"

datatype
  com = SKIP
  | Assign loc aexp ("_ := _ " 60)
  | Semi com com ("_; _" [60, 60] 10)
  | Cond bexp com com ("IF _ THEN _ ELSE _" 60)
  | While bexp com ("WHILE _ DO _" 60)
Some Fancy Type Declarations

typedeclocl loc  -- "an unspecified type of locations"

type_synonym val = nat  -- "values"

type_synonym state = "loc => val"

type_synonym aexp = "state => val"

type_synonym bexp = "state => bool"  -- "functions on states"

datatype
  com = SKIP
  | Assign loc aexp
  | Semi   com com
  | Cond   bexp com com
  | While  bexp com

new basic types

concrete syntax for commands

recursive type of commands
Notes on Type Declarations
Notes on Type Declarations

• Type synonyms merely introduce *abbreviations*. 
Notes on Type Declarations

• Type synonyms merely introduce *abbreviations*.

• Recursive data types are less general than in functional programming languages.
  • No recursion into the domain of a function.
  • Mutually recursive definitions can be tricky.
Notes on Type Declarations

- Type synonyms merely introduce *abbreviations*.
- Recursive data types are less general than in functional programming languages.
  - No recursion into the domain of a function.
  - Mutually recursive definitions can be tricky.
- Recursive types are equipped with proof methods for *induction* and case *analysis*.
Basic Constant Definitions

text{*The square of a natural number*}
definition square :: "nat => nat" where
   "square n = n*n"

text{*The concept of a prime number*}
definition prime :: "nat => bool" where
   "prime p = (1 < p ∧ (∀m. m dvd p → m = 1 ∨ m = p))"

costants
   prime :: "nat ⇒ bool"
Notes on Constant Definitions
Notes on Constant Definitions

- Basic definitions are *not* recursive.
Notes on Constant Definitions

- Basic definitions are not recursive.
- Every variable on the right-hand side must also appear on the left.
Notes on Constant Definitions

• Basic definitions are *not* recursive.
• Every variable on the right-hand side must also appear on the left.
• In proofs, definitions are *not* expanded by default!
  • Defining the constant $C$ to denote $t$ yields the theorem $C_{\text{def}}$, asserting $C=t$.
  • Abbreviations can be declared through a separate mechanism.
Lists in Isabelle
Lists in Isabelle

• We illustrate data types and functions using a reduced Isabelle theory that lacks lists.
Lists in Isabelle

• We illustrate data types and functions using a reduced Isabelle theory that lacks lists.

• The standard Isabelle environment has a comprehensive list library:

  • Functions # (cons), @ (append), map, filter, nth, take, drop, takeWhile, dropWhile, ...
  
  • Cases: (case xs of [] ⇒ [] | x#xs ⇒ ...)
  
  • Over 600 theorems!
List Induction Principle
List Induction Principle

To show $\varphi(xs)$, it suffices to show the base case and inductive step:

- $\varphi(\text{Nil})$
- $\varphi(xs) \Rightarrow \varphi(\text{Cons}(x, xs))$
List Induction Principle

To show \( \varphi(xs) \), it suffices to show the base case and inductive step:

- \( \varphi(\text{Nil}) \)
- \( \varphi(xs) \Rightarrow \varphi(\text{Cons}(x, xs)) \)

The principle of case analysis is similar, expressing that any list has one of the forms \( \text{Nil} \) or \( \text{Cons}(x, xs) \) (for some \( x \) and \( xs \)).
Proof General

theory DemoList imports Plain (*not Main, because lists are built-in*) begin

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

fun app :: "'a list => 'a list => 'a list" where
  "app Nil ys = ys"
  | "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto

proof (prove): step 0

goal (1 subgoal):
  1. app xs Nil = xs
Proof General

```
theory DemoList imports Plain (*not Main, because lists are built-in*) begin

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

fun app :: 'a list => 'a list => 'a list" where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
apply (induct xs)
apply auto

-proof (prove): step 0

goal (1 subgoal):
  1. app xs Nil = xs
```

processed material highlighted in blue
Proof General

processed material highlighted in blue

Isabelle’s output shown in a separate window
Proof General

Isabelle’s output shown in a separate window

the very start of a proof attempt

processed material highlighted in blue
Proof by Induction

theory DemoList imports Plain (*not Main, because lists are built-in*)
begin

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

fun app :: "'a list ⇒ 'a list ⇒ 'a list" where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
  apply (induct xs)
  apply auto

-proof (prove): step 1

-goal (2 subgoals):
  1. app Nil Nil = Nil
  2. ∀a xs. app xs Nil = xs ⇒ app (Cons a xs) Nil = Cons a xs
Proof by Induction

structural induction on the list xs
Proof by Induction

- Structural induction on the list \( \text{xs} \)
- Base case and inductive step
Proof by Induction

structural induction on the list $xs$

base case and inductive step

induction hypothesis
Finishing a Proof

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

fun app :: 'a list ⇒ 'a list ⇒ 'a list  where
  "app Nil ys = ys"
| "app (Cons x xs) ys = Cons x (app xs ys)"

lemma [simp]: "app xs Nil = xs"
apply (induct xs)
apply auto
done

proof (prove): step 2

goal:
No subgoals!
```
Finishing a Proof

auto proves both
Finishing a Proof

auto proves both

We must still issue "done" to register the theorem
Another Proof Attempt

```plaintext
fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

```plaintext
proof (prove): step 1

goal (2 subgoals):
1. rev (rev Nil) = Nil
2. \(a\) xs. rev (rev xs) = xs \(\Rightarrow\) rev (rev (Cons a xs)) = Cons a xs
```
Another Proof Attempt

```
fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
apply (induct xs)
apply auto
done
```

proof (prove): step 1

goal (2 subgoals):
  1. rev (rev Nil) = Nil
  2. \(\forall a \, xs. \, rev (rev xs) = xs \implies rev (rev (Cons a xs)) = Cons a xs\)
Another Proof Attempt

Can we prove both subgoals?

```haskell
fun rev where
  "rev Nil = Nil"
  "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done
```

```haskell
proof (prove): step 1

goal (2 subgoals):
  1. rev (rev Nil) = Nil
  2. \( a \), rev (rev xs) = xs \( \Rightarrow \) rev (rev (Cons a xs)) = Cons a xs
```
Stuck!

fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma rev_rev: "rev (rev xs) = xs"
apply (induct xs)
apply auto
done

proof (prove): step 2

goal (1 subgoal):
  1. \( \forall a \, xs. \, \text{rev (rev } xs) = xs \implies \text{rev (app (rev } xs) (\text{Cons } a \, \text{Nil}) = \text{Cons } a \, xs} \)
Stuck!

auto made progress but didn’t finish
Stuck!

auto made progress but didn’t finish

looks like we need a lemma relating rev and app!
Stuck Again!

```plaintext
fun rev where
 "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma [rev_rev]: "rev (rev xs) = xs"
  apply (induct xs)

proof (prove): step 2

goal (1 subgoal):
  1. ∀a xs.
     rev (app xs ys) = app (rev ys) (rev xs) ⇒
     app (app (rev ys) (rev xs)) (Cons a Nil) =
     app (rev ys) (app (rev xs) (Cons a Nil))
```

```plaintext
Wrote /Users/lp15/Dropbox/ACS/1 - Introduction/Demolist.thy
```
Stuck Again!

we dreamt up a lemma...
Stuck Again!

we dreamt up a lemma...

But it needs another lemma!
Stuck Again!

we dreamt up a lemma...

But it needs another lemma!
Stuck Again!

we dreamt up a lemma...

But it needs another lemma!
Stuck Again!

we dreamt up a lemma...

But it needs another lemma!
The Final Piece of the Jigsaw

fun rev where
  "rev Nil = Nil"
  "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  done

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)

proof (prove): step 1

goal (2 subgoals):
  1. app (app Nil ys) zs = app Nil (app ys zs)
  2. \(\forall a\) xs.
     app (app xs ys) zs = app xs (app ys zs) 

u:%% Demolist.thy 22% L20 (Isar Utoks Abbrev; Scripting )

-u-::: Demolist.thy 22% L20 (Isar Utoks Abbrev; Scripting )
goals*  Top L1 (Isar Proofstate Utoks Abbrev; )
tool-bar goto
The Finished Proof

fun rev where
  "rev Nil = Nil"
| "rev (Cons x xs) = app (rev xs) (Cons x Nil)"

lemma [simp]: "app (app xs ys) zs = app xs (app ys zs)"
  apply (induct xs)
  apply auto
  done

lemma [simp]: "rev (app xs ys) = app (rev ys) (rev xs)"
  apply (induct xs)
  apply auto
  done

lemma rev_rev: "rev (rev xs) = xs"
  apply (induct xs)
  apply auto
  done