logical verification lecture 7
2017-05-04
program extraction and prop2
overview

program extraction

program extraction: examples

verified programs: alternative approach

formulas of prop2

terminology

proofs of prop2
Bertrand Russell shows that naive set theory (or type theory) is inconsistent:

\[ \{x \mid x \not\in x\} \in \{x \mid x \not\in x\} \]
some answers: three schools

Hilbert: **formalism**, leads eventually to ZFC set theory

Russell: **logicism**, leads eventually to an early version of type theory

Brouwer, Heyting, Bishop: **intuitionism**, rejects excluded middle
Brouwer–Heyting–Kolmogorov interpretation

⊥ does not exist

\( A \rightarrow B \) maps proofs of \( A \) to proofs of \( B \)

\( A \land B \) proof of \( A \) and proof of \( B \)

\( A \lor B \) proof of \( A \) or a proof of \( B \)

\( \forall x. P(x) \) maps \( x \) to a proof of \( P(x) \)

\( \exists x. P(x) \) object \( a \) with proof of \( P(a) \)

proof of existence corresponds to constructing an example
an intuitionistic (constructive) proof

corresponds to an executable algorithm
constructive functional programming

- program specification
- constructive proof of existence
- automatically generated functional program
program specification: example

the correctness proof of the specification

\[ \forall l : \text{natlist}. \exists l' : \text{natlist}. \text{permutation}(l, l') \land \text{sorted}(l') \]

yields a program (function) from natlist to natlist
program specification: general pattern

∀x : A. P(x) → ∃y : B. Q(x, y)

- A  input type
- B  output type
- P(x)  precondition
- Q(x, y)  input/output behaviour

the correctness proof yields a program from A to B
program extraction in Coq

Coq proof in type theory gives
functional program in OCaml or Haskell or Scheme
program extraction in Coq

is “almost” the identity function but

- other typing system
- information from Prop is erased
existential quantification in Prop

inductive type:

\[
\text{Inductive } \text{ex} \ (A : \text{Type}) \ (P : A \to \text{Prop}) : \text{Prop} := \\
\quad \text{ex_intro} : \forall x : A, P x \to \text{ex } P
\]

syntax:

\[
\text{exists } x : A, P x.
\]
existential quantification in Set

inductive type:

\[
\text{Inductive } \text{sig } (A : \text{Set}) \ (P : A \to \text{Prop}) : \text{Set} := \\
\text{exist} : \forall x : A, P x \to \text{sig } P
\]

syntax:

\[\{x:A \mid P \ x\}\]
for program extraction

use existential quantification in Set
successor: existence proof and extracted program

specification:

Theorem successor :
    forall n:nat, \{m:nat | m = S n\}.

extracted program:

let successor n =
    S n
Theorem predecessor :
   forall n:nat, ~(n = 0) -> {m:nat | S m = n}.

let rec predecessor = function
 | 0 -> assert false (* absurd case *)
 | S n0 -> n0
insertion sort: existence proof

Theorem Sort :
  \forall l : \text{natlist},
  \{ l' : \text{natlist} | \text{permutation} l l' \land \text{sorted} l' \}. 
insertion sort: predicate permutation

Inductive permutation : natlist -> natlist -> Prop :=
  | permutation_nil : permutation nil nil
  | permutation_cons :
    forall (n : nat) (l l' l'' : natlist),
    permutation l l' ->
    inserted n l' l'' ->
    permutation (cons n l) l''.
insertion sort: predicate inserted

Inductive inserted (n : nat) :
  : natlist -> natlist -> Prop :=
  | inserted_front :
      forall l : natlist, inserted n l (cons n l)
  | inserted_cons :
      forall (m : nat) (l l’ : natlist),
      inserted n l l’ ->
      inserted n (cons m l) (cons m l’).

le: family of inductive predicates

Inductive le (n:nat) : nat -> Prop :=
| le_n : le n n
| le_S : forall m:nat , le n m -> le n (S m).

le_ind
: forall (n : nat) (P : nat -> Prop),
P n ->
 (forall m : nat, le n m -> P m -> P (S m)) ->
forall n0 : nat, le n n0 -> P n0
le: examples

le_n 0 : le 0 0 : Prop
le_n 7 : le 7 7 : Prop
le_S 0 0 (le_n 0) : le 0 1 : Prop
le_S 0 1 (le_S 0 0 (le_n 0)) : le 0 2 : Prop
Inductive sorted : natlist -> Prop :=
  | sorted0 : sorted nil
  | sorted1 : forall n:nat , sorted (cons n nil)
  | sorted2 : forall n h:nat , forall t:natlist,
    le n h ->
    sorted (cons h t) ->
    sorted (cons n (cons h t)).
Leibniz equality

two terms are equal if they have the same properties

Inductive eq (A : Type) (x : A) : A -> Prop :=
  refl_equal : x = x

eq_ind
  : forall (A : Type) (x : A) (P : A -> Prop),
    P x -> forall y : A, x = y -> P y
verified programs: two approaches

• correctness proofs
  from program to proof

• program extraction
  from proof to program
correctness proofs: Hoare logic

imperative program
\[\Rightarrow\]
annotated imperative program
\[\Rightarrow\]
proof obligations
define a function mirror
and prove its correctness:

Theorem Mirrored_mirror :
forall t : bintree,
Mirrored t (mirror t).
mirror: program extraction

prove the specification correct
and extract a program from it

Theorem Mirror : forall t : bintree,
   {t’ : bintree | Mirrored t t’}.

summarizing the two approaches

- **specification**
  Inductive Mirrored

- **approach 1: implementation**
  Fixpoint mirror

- **approach 1: correctness**
  Theorem Mirrored mirror

- **approach 2: program extracted from existence proof**
  Theorem Mirror
logics and type theory

1st-order minimal propositional logic $\leftrightarrow$ simple type theory

1st-order minimal predicate logic $\leftrightarrow$ dependent type theory

2nd-order minimal propositional logic $\leftrightarrow$ polymorphic type theory
formulas of prop1 (already seen)

\[a \ b \ c \ p \ q\]

\[A \rightarrow B\]

\[\perp\]

\[\top\]

\[A \land B\]

\[A \lor B\]
formulas of pred1 (already seen)

(using terms)

\[ a(\ldots) \quad b(\ldots) \quad c(\ldots) \quad p(\ldots) \quad q(\ldots) \]
\[ A \rightarrow B \]
\[ \forall x. A \]
\[ \bot \]
\[ \top \]
\[ A \land B \]
\[ A \lor B \]
\[ \exists x. A \]
formulas of prop2 (new)

\[ a \ b \ c \ p \ q \]
\[ A \rightarrow B \]
\[ \forall a. A \]
\[ \bot \]
\[ \top \]
\[ A \land B \]
\[ A \lor B \]
\[ \exists a. A \]
examples

in prop1:
\[ a \rightarrow a \]

in pred1:
\[ \forall x. a(x) \rightarrow a(x) \]

in prop2:
\[ \forall a. a \rightarrow a \]

for every proposition, that proposition implies itself
higher-order

**first order:**
object

**second order:**
set of first-order objects
predicate on objects
function from objects to objects

**third order:**
set of second-order objects
predicate on predicates on objects
functions from second order objects
higher-order logic

first-order:
quantification over variables of order 1
\[ a \rightarrow a \]
\[ \forall x. a(x) \rightarrow a(x) \]

second-order:
quantification over variables of order 2
\[ \forall a. a \rightarrow a \]
\[ \forall a. \forall x. a(x) \rightarrow a(x) \]
\[ \forall f. \forall x. a(f(x)) \rightarrow a(f(x)) \]

third-order:
quantification over variables of order 3
\[ \forall b. \forall f. b(f) \rightarrow \forall x. a(f(x)) \]

quantify over predicates gives pred2
same without terms gives prop2
second-order predicate logic: example

induction principle for natural numbers

\[ \forall a. a(0) \rightarrow (\forall m. a(m) \rightarrow a(S(m))) \rightarrow \forall n. a(n) \]

- \( m \) 1st order variable
- \( n \) 1st order variable
- \( 0 \) 1st order constant
- \( a \) 2nd order variable
- \( S \) 2nd order constant (or 1st order function)
there exists a sorting function

$$\exists f : \text{natlist} \rightarrow \text{natlist}. \forall l : \text{natlist}. \text{sorted}(f(l)) \land \text{permutation}(l, f(l))$$

- $f$: 2nd order variable
- $l$: 1st order variable
- $\text{sorted}$: 2nd order constant (or 1st order function)
- $\text{permutation}$: 2nd order constant (or 1st order function)
<table>
<thead>
<tr>
<th>prop2</th>
<th>pred2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\forall a. a \rightarrow a$</td>
<td>$\forall p. \forall x. p(x) \rightarrow p(x)$</td>
</tr>
<tr>
<td>prop1</td>
<td>pred1</td>
</tr>
<tr>
<td>$a \rightarrow a$</td>
<td>$\forall x. p(x) \rightarrow p(x)$</td>
</tr>
</tbody>
</table>
proof rules for prop2

\begin{align*}
\text{introduction rules} & \quad \text{elimination rules} \\
I \top & \quad E \bot \\
I[x] \rightarrow & \quad E \rightarrow \\
I \land & \quad EI \land, Er \land \\
I \lor, Ir \lor & \quad E \lor \\
I \forall & \quad E \forall \\
I \exists & \quad E \exists
\end{align*}
universal quantification for prop2

∀ introduction:

\[
\begin{align*}
A & \quad \text{I}\\
\forall a. A & \quad \text{I}\forall
\end{align*}
\]

variable condition: \( a \) not free in any open assumption
check: variable does not occur in any of the available assumptions

∀ elimination:

\[
\begin{align*}
\forall a. A & \quad E\forall\\
A[a := B] & \quad E\forall
\end{align*}
\]
existential quantification for prop2

∃ introduction:

\[
\frac{A[a := B]}{∃a. A} \quad I∃
\]

∃ elimination:

\[
\frac{∃a. A \quad ∀a. A \rightarrow B}{B} \quad E∃
\]

variable condition: \( a \) not free in \( B \)
check: variable does not occur in the conclusion
examples of tautologies

- $\forall b. b \rightarrow a$
- $a \rightarrow \forall b. (b \rightarrow a)$
- $a \rightarrow \forall b. ((a \rightarrow b) \rightarrow b)$
- $\exists b. a \rightarrow a$
- $\exists b. ((a \rightarrow b) \lor (b \rightarrow a))$
examples of non-tautological formulas

• $a \rightarrow (\forall a. a)$

• $p(x) \rightarrow (\forall x. p(x))$

• $(\exists a. a) \rightarrow a$

• $\forall a. \forall b. (a \rightarrow b) \lor (b \rightarrow a)$
  (classical logic needed)
minimal prop2: detour

introduction rule for a connective

immediately followed by an

elimination rule for the same connective
elimination of an implication detour (as in prop1)

\[ \frac{\cdot}{\bar{B}} \]
\[ \frac{A \rightarrow B}{B} \]
\[ \text{I}[x] \rightarrow \]
\[ \frac{\cdot}{\bar{A}} \]
\[ E \rightarrow \]

is replaced by

\[ \cdot \]
\[ \frac{\cdot}{\bar{B}} \]

where every occurrence of the assumption $A^x$ is replaced by the proof

\[ \cdot \]
\[ \frac{\cdot}{\bar{A}} \]
elimination of an universal quantification detour

(similar to pred1)

\[
\frac{B}{\forall a. B} \quad \forall \forall
\]

\[
\frac{}{B[a := A]} \quad E\forall
\]

\[
\rightarrow \quad B[a := A]
\]

everywhere \(a\) is replaced by \(A\)