F*: Prove your Programs

Chantal Keller

June, 17th 2014

Motivation


your choice below and voting for individual
candidates in any other party.
REPUBLICAN
DEMOCRATIC
GREEN
LIBERTARIAN
PRESIDENTIAL ELECTORS
Vote for the candidates of one Party for the President and Vice-President or insert the names of candidates.
MITT ROMNEY, President
PAUL RYAN, Vice-President
REPUBLICAN
BARACK OBAMA, President
JOE BIDEN, Vice-President
DEMOCRATIC
JILL STEIN, President
CHERI HONKALA, Vice-President
GREEN
GARY JOHNSON, President
JAMES P GRAY, Vice-President
LIBERTARIAN
Write-In
01 / 02


Rapid
A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) + 00010E36. The current application will be terminated.
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will lose any unsaved information in all applications.
Press any key to continue _

Some solutions

Human:

- readable code on the long term
- clear specifications
- paper proof
- appropriate programming language
- appropriate development environment (text editor, version control system. . .)

Computer-aided:

- unit tests
- features of the programming languages: typing, warnings. . .
- formal methods: mathematical specifications + proofs or/and large tests

Some solutions

Human:

- readable code on the long term
- clear specifications
- paper proof
- appropriate programming language
- appropriate development environment (text editor, version control system. . .)

Computer-aided:

- unit tests
- features of the programming languages: typing, warnings. . .
- formal methods: mathematical specifications + proofs or/and large tests

Formal methods

Two kinds of properties:

- "never goes wrong": do not raise exceptions at runtime, no illegal memory access, termination. . .
- functional soundness: match the specifications

Formal methods

Two kinds of properties:

- “never goes wrong”: do not raise exceptions at runtime, no illegal memory access, termination. . .
- functional soundness: match the specifications

Deductive verification

program

3 steps:

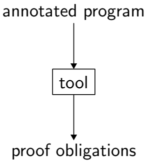
Deductive verification

annotated program

3 steps:

1. annotations: mathematical specifications

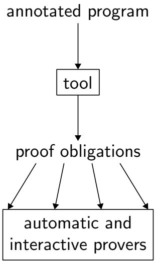
Deductive verification



3 steps:

1. annotations: mathematical specifications
2. generation of proof obligations

Deductive verification



3 steps:

1. annotations: mathematical specifications
2. generation of proof obligations
3. prove them

The F* language

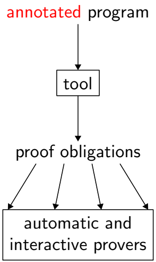
F*:

- functional programming language: programs are functions...
- higher-order aspects: ...that can manipulate functions
- side effects: input/output, arrays...

The 3 steps:

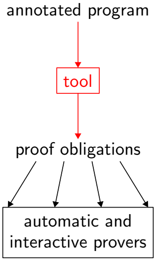
- *refinement types* to express specifications
- *weakest-preconditions calculus* to transform specifications + code into a formula to check
- *transformation of the formula* to be passed to automated theorem provers

Step 1: refinement types



- demo: toy example
- large application: miTLS

Step 2: weakest-preconditions calculus



Main idea

```
val f: x:T₁{Pre(x)} → r:T₂{Post(x,r)}
let rec f x = ...
```

Main idea

logical formulas

```
val f: x:T₁{Pre(x)} → r:T₂{Post(x,r)}
let rec f x = ...
```

Main idea

logical formulas

```
val f: x:T₁{Pre(x)} → r:T₂{Post(x,r)}
let rec f x = ...
```

- given the code for f and the postcondition
- compute the weakest precondition WP(x) that implies the postcondition after running f:

$$\forall\, x\ r.\ WP(x) \Rightarrow Post(x,r)$$

Main idea

logical formulas

```
val f: x:T₁{Pre(x)} → r:T₂{Post(x,r)}
let rec f x = ...
```

- given the code for f and the postcondition
- compute the weakest precondition WP(x) that implies the postcondition after running f:

$$\forall\ x\ r.\ WP(x) \Rightarrow Post(x,r)$$

(Next step: show that the given precondition implies the computed one.

$$\forall x.\ Pre(x) \Rightarrow WP(x))$$

Running example

```
val max: x:int → y:int →
        z:int{z ⩾ x ∧ z ⩾ y ∧ (z = x ∨ z = y)}
let max x y = if x > y then x else y
```

- Precondition: Pre(x,y) = **true**
- Postcondition: Post(x,y,z) $= z \geqslant x \wedge z \geqslant y \wedge (z = x \vee z = y)$

Running example

```
val max: x:int → y:int →
         z:int{z ⩾ x ∧ z ⩾ y ∧ (z = x ∨ z = y)}
let max x y = if x > y then x else y
```

- Precondition: Pre(x,y) = **true**
- Postcondition: Post(x,y,z) = $z \geqslant x$

Running example

```
val max: x:int → y:int →
        z:int{z ⩾ x ∧ z ⩾ y ∧ (z = x ∨ z = y)}
let max x y = if x > y then x else y
```

- Precondition: Pre(x,y) = **true**
- Postcondition: Post(x,y,z) = $z \geqslant x$

How to show that (**if** $x > y$ **then** x **else** y) $\geqslant x$?

Running example

```
val max: x:int → y:int →
         z:int{z ⩾ x ∧ z ⩾ y ∧ (z = x ∨ z = y)}
let max x y = if x > y then x else y
```

- Precondition: Pre(x,y) = **true**
- Postcondition: Post(x,y,z) = $z \geqslant x$

How to show that (**if** x > y **then** x **else** y) ⩾ x?

- must be true in both branches, knowing the result of the test

Running example

```
val max: x:int → y:int →
         z:int{z ⩾ x ∧ z ⩾ y ∧ (z = x ∨ z = y)}
let max x y = if x > y then x else y
```

- Precondition: Pre(x,y) $=$ **true**
- Postcondition: Post(x,y,z) $= z \geqslant x$

How to show that (**if** $x > y$ **then** x **else** y) $\geqslant x$?

- must be true in both branches, knowing the result of the test
- left: $x > y \Rightarrow x \geqslant x$

Running example

```
val max: x:int → y:int →
          z:int{z ⩾ x ∧ z ⩾ y ∧ (z = x ∨ z = y)}
let max x y = if x > y then x else y
```

- Precondition: Pre(x,y) = **true**
- Postcondition: Post(x,y,z) = $z \geqslant x$

How to show that (**if** $x > y$ **then** x **else** y) $\geqslant x$?

- must be true in both branches, knowing the result of the test
- left: $x > y \Rightarrow x \geqslant x$
- right: $x \leqslant y \Rightarrow y \geqslant x$

Running example

```
val max: x:int → y:int →
         z:int{z ⩾ x ∧ z ⩾ y ∧ (z = x ∨ z = y)}
let max x y = if x > y then x else y
```

- Precondition: Pre(x,y) = **true**
- Postcondition: Post(x,y,z) = $z \geqslant x$

How to show that (**if** $x > y$ **then** x **else** y) $\geqslant x$?

- must be true in both branches, knowing the result of the test
- left: $x > y \Rightarrow x \geqslant x$
- right: $x \leqslant y \Rightarrow y \geqslant x$

The weakest precondition is: $(x > y \Rightarrow x \geqslant x) \wedge (x \leqslant y \Rightarrow y \geqslant x)$

In general

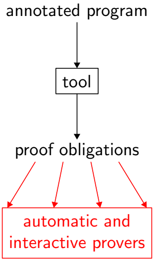
Proceed step by step on the code:

- here we have applied the rule:

  $$\mathrm{WP}(\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2,\ P) = (b \Rightarrow \mathrm{WP}(e_1,\ P)) \wedge (\neg\ b \Rightarrow \mathrm{WP}(e_2,\ P))$$

- other rules for the other constructions of the language (loops, assignments, **let rec**. . .)

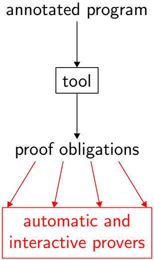
Step 3: prove the final formula



Check that $\forall x.\ Pre(x) \Rightarrow WP(x)$:

$$\textbf{true} \Rightarrow (x > y \Rightarrow x \geqslant x) \land (x \leqslant y \Rightarrow y \geqslant x)$$

Step 3: prove the final formula



Check that $\forall x.\ Pre(x) \Rightarrow WP(x)$:

$$\textbf{true} \Rightarrow (x > y \Rightarrow x \geqslant x) \wedge (x \leqslant y \Rightarrow y \geqslant x) \checkmark$$

In general

Automatically:

- SMT solvers (Z3, Alt-Ergo, veriT, CVC3, . . .): theory reasoning (accesses in arrays, arithmetic, . . .)
- first-order provers (Vampire, E-prover, . . .): quantifiers

Interactively:

- interactive theorem provers (Coq, Isabelle, PVS, . . .): expressivity and safety

Current research

F*:

- higher-order aspects: gives higher-order goal
- functions in the logic must be total: automatically guess totality
- increase confidence in the final check: automatically re-check SMT solver's answers in proof assistants (SMTCoq)
- provide back-ends for various languages (JavaScript, OCaml...)

Other topics:

- make these software more accessible
- increase expressivity and automation in the final check
- distributed programs

Recommendations

Correctness w.r.t specifications:

- specs might not be what you expect (demo)
- specs might be hard to express (eg. user interface)

Time consuming, but:

- very strong safety
- fun!

Prove your programs

Many different tools for formal methods:

- deductive verification
- interactive theorem provers
- software synthesis
- model checking
- abstract interpretation
- ...

Enter a bug-free world!