F*: Prove your Programs

Chantal Keller

June, 17th 2014
Motivation
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

3 steps:

- Annotations: mathematical specifications
- Generation of proof obligations
- Prove them
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

```ocaml
val f : \(x:T_1\{\text{Pre}(x)\} \rightarrow r:T_2\{\text{Post}(x,r)\}
let rec f x = ...
```
Main idea

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

\[
\text{val } f : \ x : T_1 \{ \text{Pre}(x) \} \rightarrow \ r : T_2 \{ \text{Post}(x, r) \} \\
\text{let rec } f \ x = \ldots
\]

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

\[
\forall x \ r. \ WP(x) \Rightarrow \text{Post}(x,r)
\]
Main idea

\[ \text{val } f : x : T_1 \{ \text{Pre}(x) \} \rightarrow r : T_2 \{ \text{Post}(x, r) \} \]

\text{let rec } f \ x = \ldots \]

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

\[ \forall x \ r. \ WP(x) \Rightarrow \text{Post}(x,r) \]

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

\[ \forall x. \ \text{Pre}(x) \Rightarrow WP(x) \]
Running example

```f
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 ≥ x ∧ z ≥ y ∧ (z = x ∨ z = y)
Running example

```f
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: \( \text{Pre}(x,y) = \text{true} \)

- Postcondition: \( \text{Post}(x,y,z) = z ⩾ x \)
Running example

\[
\text{val max: } x: \text{int} \rightarrow y: \text{int} \rightarrow \text{int}\{z \geq x \land z \geq y \land (z = x \lor z = y)\} \\
\text{let max } x \ y = \text{if } x > y \ \text{then } x \ \text{else } y
\]

- Precondition: \(\text{Pre}(x,y) = \text{true}\)
- Postcondition: \(\text{Post}(x,y,z) = z \geq x\)

How to show that \((\text{if } x > y \ \text{then } x \ \text{else } y) \geq x\)?
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 ≥ 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

\[
\text{val max}: \ x:\text{int} \to y:\text{int} \to \\
\quad z:\text{int}\{z \geq x \land z \geq y \land (z = x \lor z = y)\}
\]

\[
\text{let max } x \ y = \textbf{if } x > y \ \textbf{then } x \ \textbf{else } y
\]

- Precondition: \(\text{Pre}(x,y) = \text{true}\)
- Postcondition: \(\text{Post}(x,y,z) = z \geq x\)

How to show that \((\textbf{if } x > y \ \textbf{then } x \ \textbf{else } y) \geq x?\)

- must be true in both branches, knowing the result of the test
- left: \(x > y \Rightarrow x \geq x\)
Running example

```plaintext
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 ≥ 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
  - left: x > y ⇒ x ≥ x
  - right: x ≤ y ⇒ y ≥ x
Running example

\[
\text{val } \text{max} : \ x : \text{int} \rightarrow y : \text{int} \rightarrow \\
\quad \text{z : int} \{ z \geq x \wedge z \geq y \wedge (z = x \lor z = y) \}
\]

\[
\text{let } \text{max } x \ y = \text{if } x > y \ \text{then } x \ \text{else } y
\]

- **Precondition:** \( \text{Pre}(x, y) = \text{true} \)
- **Postcondition:** \( \text{Post}(x, y, z) = z \geq x \)

**How to show that** \( (\text{if } x > y \ \text{then } x \ \text{else } y) \geq x \)?

- **must be true in both branches, knowing the result of the test**
  - **left:** \( x > y \Rightarrow x \geq x \)
  - **right:** \( x \leq y \Rightarrow y \geq x \)

**The weakest precondition is:** \( (x > y \Rightarrow x \geq x) \wedge (x \leq y \Rightarrow y \geq x) \)
In general

Proceed step by step on the code:

- here we have applied the rule:
  \[
  \text{WP}(\text{if } b \text{ then } e_1 \text{ else } e_2, P) = \\
  (b \Rightarrow \text{WP}(e_1, P)) \land (\neg b \Rightarrow \text{WP}(e_2, P))
  \]

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

F*: Prove your Programs

Chantal Keller
Step 3: prove the final formula

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

$$\text{true} \Rightarrow (x > y \Rightarrow x \geq x) \land (x \leq y \Rightarrow y \geq x)$$
Step 3: prove the final formula

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

$$\text{true} \Rightarrow (x > y \Rightarrow x \geq x) \land (x \leq y \Rightarrow y \geq x) \checkmark$$
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!