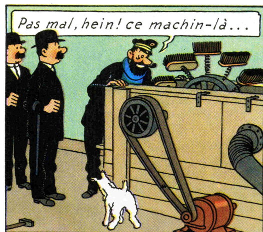
# F*: a general purpose language for program verification

Chantal Keller
Joint work with lots of people

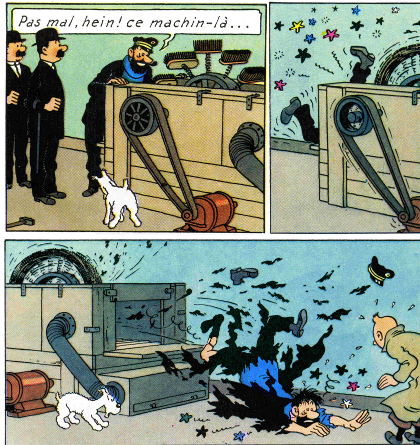
February, 2nd 2015

Microsoft Research

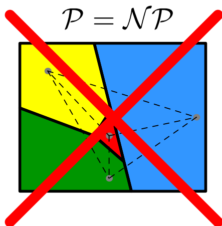
# Trust automatic devices



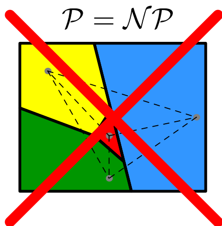
# Trust automatic devices

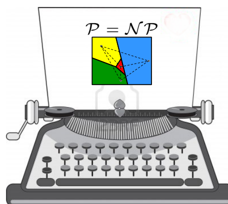


# A machine to certify other machines



# A machine to certify other machines



# Main goal

To build and deploy systems that are provably secure, end-to-end

- an ML-like programming language
- more expressive type system, that allows to express arbitrary properties about programs
- automatic proof of these properties
- fully-abstract code generation and deployment to various platforms

# Main goal

To build and deploy systems that are provably secure, end-to-end

- an ML-like programming language
  higher-order, stateful, non-terminating programs, polymorphism, datatypes, ...
- more expressive type system, that allows to express arbitrary properties about programs
- automatic proof of these properties
- fully-abstract code generation and deployment to various platforms

# Main goal

To build and deploy systems that are provably secure, end-to-end

- an ML-like programming language

  higher-order, stateful, non-terminating programs, polymorphism, datatypes, ...
- more expressive type system, that allows to express arbitrary properties about programs

  functional correctness, policies, ...
- automatic proof of these properties
- fully-abstract code generation and deployment to various platforms

# Main goal

To build and deploy systems that are provably secure, end-to-end

- an ML-like programming language

  higher-order, stateful, non-terminating programs, polymorphism, datatypes, . . .
- more expressive type system, that allows to express arbitrary properties about programs

  functional correctness, policies, . . .
- automatic proof of these properties

  verification condition generation discharged by SMT solvers
- fully-abstract code generation and deployment to various platforms

# Main goal

To build and deploy systems that are provably secure, end-to-end

- an ML-like programming language

  higher-order, stateful, non-terminating programs, polymorphism, datatypes, . . .
- more expressive type system, that allows to express arbitrary properties about programs

  functional correctness, policies, . . .
- automatic proof of these properties

  verification condition generation discharged by SMT solvers
- fully-abstract code generation and deployment to various platforms

  OCaml, JavaScript, F#, . . .

# Examples

```
val f : int → int
let f x = x + 1
```

# Examples

```
val f: x:int → y:int{y > x}
let f x = x + 1
```

# Examples

```
val f: x: int → y:int{y > x}
let f x = x + 1


val quickSort: f:(a → a → Tot bool){total_order a f}
    → l:list a
    → Tot (m:list a{sorted f m ∧ ∀ x, count x l = count x m})
let rec quickSort f = ...
```

# Examples

```
val f: x:int → y:int{y > x}
let f x = x + 1


val quickSort: f:(a → a → Tot bool){total_order a f}
    → l:list a
    → Tot (m:list a{sorted f m ∧ ∀ x, count x l = count x m})
let rec quickSort f = ...


val counter: unit → ST (x:int{x ⩾ 0})
let counter =
  let c = ref (−1) in
  fun () → c := !c+1; !c
```

# Examples

```
val f: x:int → y:int{y > x}
let f x = x + 1
```

```
val quickSort: f:(a → a → Tot bool){total_order a f}
    → l:list a
    → Tot (m:list a{sorted f m ∧ ∀ x, count x l = count x m})
let rec quickSort f = ...
```

```
val counter: unit → ST (x:int{x ⩾ 0})
let counter =
  let c = ref (−1) in
  fun () → c := !c+1; !c
```

↪ static and complete verification

# More concrete example: policies

## On the server side:

```
let canWrite (d: directory ) =
  d = ''/tmp''
```

```
let canRead (d: directory ) =
  canWrite d || d=''/public ''
```

```
assume val read  : f:filename{canRead (dir f)} → string
assume val write : f:filename{canWrite (dir f)} → string → unit
```

# More concrete example: policies

## On the server side:

```
let canWrite (d: directory) =
  d = "/tmp"
```

```
let canRead (d: directory) =
  canWrite d || d="/public"
```

```
assume val read  : f:filename{canRead (dir f)}  → string
assume val write : f:filename{canWrite (dir f)} → string → unit
```

## On the client side:

```
let niceClient () =
  let v1 = read "/tmp/foo.txt" in
  let v2 = read "/public/readme" in
  write "/tmp/bar.txt" "hello!"
```

```
let evilClient () =
  let v1 = read "/etc/passwd" in
  write "/tmp/bar.txt" "ha ha!"
```

# Deductive verification in F*

program

4 steps:

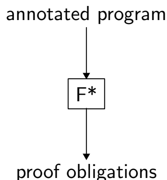
# Deductive verification in F*

annotated program

4 steps:

1. annotations: mathematical specifications

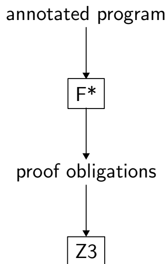
# Deductive verification in F*




4 steps:

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

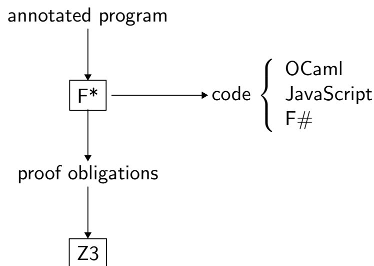
# Deductive verification in F*




4 steps:

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

# Deductive verification in F*




4 steps:

1. annotations: mathematical specifications
2. automatic generation of proof obligations
3. automatic proof of them
4. automatic code generation for deployment

# Conclusion

## Real-world examples

- miTLS: certification of the TLS protocol
- correctness of the F* type-checker itself

$\hookrightarrow$ relies on the fact that F* is a general-purpose language

# Conclusion

## Real-world examples

- miTLS: certification of the TLS protocol
- correctness of the F* type-checker itself

↪ relies on the fact that F* is a general-purpose language