Verifying Relations Between F* Programs

Nik Swamy

OPLSS 19

Relational Verification

Relating multiple programs,
Or multiple executions of a single program:

- Program equivalence, e.g., correctness of optimizations
- Program refinement
- Security properties, including hyper-properties like non-interference
- . . .

This talk based on **A Monadic Framework for Relational Verification, CPP 2018**

Effects: A Central Difficulty in Relational Verification

- Relations between pure programs are ... relatively easy.
- But, how to even state relations between effectful programs?

Effects: A Central Difficulty in Relational Verification

- Relations between pure programs are ... relatively easy.
- But, how to even state relations between effectful programs?
- Many custom logics and tools to support stating and proving relations between effectul programs.
  - Benton (Relational Hoare Logic),
  - Barthe at al (Probabilistic RHL, EasyCrypt),
  - Type systems for information flow control (many)
  - . . .

Main Idea of this Work

(dead simple)

- *Program* effectful computations in an abstract, monadic style
  - Abstraction enables effects to be compiled primitively, e.g., state with destructive updates

Main Idea of this Work

(dead simple)

- *Program* effectful computations in an abstract, monadic style
    - Abstraction enables effects to be compiled primitively, e.g., state with destructive updates
- *Reason* about effectful computations by revealing their pure, monadic representations
    - Reduce relating effectful computations to relating pure functions.

A basic example

Consider proving these two stateful, ML programs equivalent:

```
let rec sum_up r lo hi =
  if lo ≠ hi then (r := r + lo ; sum_up r (lo+1) hi)

let rec sum_down r lo hi =
  if lo ≠ hi then (r := r + hi ; sum_down r lo (hi−1))
```

- Both programs add the same value to the reference r
- But they compute it in a different order

A basic example: Attempt 1

Consider proving these two stateful, ML programs equivalent:

```
let rec sum_up r lo hi =
  if lo ≠ hi then (r := r + lo ; sum_up r (lo+1) hi)

let rec sum_down r lo hi =
  if lo ≠ hi then (r := r + hi ; sum_down r lo (hi−1))
```

Separate unary, functional correctness proofs

- Prove them functionally correct separately, e.g., using some kind of Floyd-Hoare logic
- And prove that their pure functional specs are equivalent

A basic example: Attempt 1

Consider proving these two stateful, ML programs equivalent:

```
let rec sum_up r lo hi =
  if lo ≠ hi then (r := r + lo ; sum_up r (lo+1) hi)

let rec sum_down r lo hi =
  if lo ≠ hi then (r := r + hi ; sum_down r lo (hi−1))
```

Separate unary, functional correctness proofs

- Prove them functionally correct separately, e.g., using some kind of Floyd-Hoare logic
- And prove that their pure functional specs are equivalent
- But, this is tedious: required writing separate functional specs
- And this style of proof may not always be possible
    - Not every hyper-property can be expressed as a collection of unary properties

A basic example: Attempt 2

Consider proving these two stateful, ML programs equivalent:

```
let rec sum_up r lo hi =
  if lo ≠ hi then (r := r + lo ; sum_up r (lo+1) hi)

let rec sum_down r lo hi =
  if lo ≠ hi then (r := r + hi ; sum_down r lo (hi−1))
```

Relate the monadic representations of the stateful computations

- Prove `sum_up r lo hi` $\sim$ `sum_dn r lo hi`
- Where $c_0 \sim c_1$ relates `mem -> a * mem` pure computations
  - i.e., relating the monadic representations of c_0, c_1.

Applying this approach to relational verification in F*

A recipe with 5 main ingredients:

Applying this approach to relational verification in F*

A recipe with 5 main ingredients:

- Effectful programming with abstract, monadic computations, extracted to efficient imperative code in OCaml, F#, C

Applying this approach to relational verification in F*

A recipe with 5 main ingredients:

- Effectful programming with abstract, monadic computations, extracted to efficient imperative code in OCaml, F#, C
- A (unary) Hoare-style program logic

Applying this approach to relational verification in F*

A recipe with 5 main ingredients:

- Effectful programming with abstract, monadic computations, extracted to efficient imperative code in OCaml, F#, C
- A (unary) Hoare-style program logic
- Monadic reification, making effectul computations pure

Applying this approach to relational verification in F*

A recipe with 5 main ingredients:

- Effectful programming with abstract, monadic computations, extracted to efficient imperative code in OCaml, F#, C
- A (unary) Hoare-style program logic
- Monadic reification, making effectul computations pure
- A rich dependently typed logic, well-suited to reasoning by computation about pure computations

Applying this approach to relational verification in F*

A recipe with 5 main ingredients:

- Effectful programming with abstract, monadic computations, extracted to efficient imperative code in OCaml, F#, C
- A (unary) Hoare-style program logic
- Monadic reification, making effectul computations pure
- A rich dependently typed logic, well-suited to reasoning by computation about pure computations
- Semi-automated proofs, by encoding to SMT

These ingredients are not unique to F*

Dependent types Monad-based effects	Hoare logic, imperative programs SMT-based automation
Coq, Agda Lean, Idris Isabelle (HOL)	Dafny, Boogie, Vcc FramaC, Why3, Verifast, ...

But their combination may be.

Back to our running example

```
let rec sum_up r lo hi =
  if lo ≠ hi then (r := r + lo ; sum_up r (lo+1) hi)

let rec sum_down r lo hi =
  if lo ≠ hi then (r := r + hi ; sum_down r lo (hi−1))
```

We want to show that on any initial memory, the programs `sum_up`, and `sum_dn` results in related memories.

State as a monad

We start from a monadic presentation of state

```
type st (mem:Type) (a:Type) = mem → Tot (a ∗ mem)

let return (x:a) : st mem a = λh → (x, h)

let bind (c0 : st mem a) (f: a → st mem b) : st mem b =
  λh0 → let x, h1 = c0 h0 in f x h1

let get () : st mem mem = λh → (h,h)

let put (h:mem) : st mem unit = λh0 → ((), h)
```

State as a monad

We start from a monadic presentation of state

```
type st (mem:Type) (a:Type) = mem → Tot (a ∗ mem)

let return (x:a) : st mem a = λh → (x, h)

let bind (c0 : st mem a) (f: a → st mem b) : st mem b =
  λh0 → let x, h1 = c0 h0 in f x h1

let get () : st mem mem = λh → (h,h)

let put (h:mem) : st mem unit = λh0 → ((), h)
```

... and generate an abstract effect which can be implemented primitively !

```
total new_effect { STATE : a:Type → Effect
  with repr = st heap ; ... }
```

Unary Hoare Logic

$$\Gamma \vdash \{\text{pre}\}\,\text{code}\,\{\text{post}\}$$

$$\Gamma \vdash \text{code} : \text{ST a (requires pre) (ensures post)}$$

For stateful code :

- pre : h0:heap $\rightarrow$ prop
- post : h0:heap $\rightarrow$ result:a $\rightarrow$ h1:heap $\rightarrow$ prop

Unary Hoare Logic

$$\Gamma \vdash \{\text{pre}\}\,\text{code}\,\{\text{post}\}$$

$$\Gamma \vdash \text{code} : \text{ST a (requires pre) (ensures post)}$$

For stateful code :

- pre : h0:heap $\rightarrow$ prop
- post : h0:heap $\rightarrow$ result:a $\rightarrow$ h1:heap $\rightarrow$ prop

We can intrisically specify our programs :

```
val sum_up : r:ref int → lo:int → hi:int → ST unit
        (requires λh0 → lo ≤ hi ∧ h0 `contains' r)
        (ensures λh0 () h1 → h1 `contains' r)

let rec sum_up r lo hi =
  if lo ≠ hi then (r := r + lo ; sum_up r (lo+1) hi)
```

Reifying effectful computations, for logical reasoning only

(the main idea)

Monadic reification, an idea from Filinski, but here only for logical reasoning

$$\begin{array}{l}\Gamma \vdash e : \mathsf{ST}\ \mathsf{a}\ (\mathsf{requires}\ \mathsf{pre})\ (\mathsf{ensures}\ \mathsf{post}) \\ \Longrightarrow \\ \Gamma \vdash \mathsf{reify}\ e : \\ \qquad \mathsf{h0} : \mathsf{heap}\{\mathsf{pre}\ \mathsf{h0}\} \rightarrow \mathsf{r}{:}(\mathsf{a} * \mathsf{heap})\{\mathsf{post}\ \mathsf{h0}\ (\mathsf{fst}\ \mathsf{r})\ (\mathsf{snd}\ \mathsf{r})\}\end{array}$$

Reification expose the monadic *model* of an effect in specification

Reifying effectful computations, for logical reasoning only

(the main idea)

Monadic reification, an idea from Filinski, but here only for logical reasoning

$$\Gamma \vdash e : \text{STATE a (requires pre) (ensures post)}$$

$$\Longrightarrow$$

$$\Gamma \vdash \text{reify } e : \text{GHOST}$$

$$\text{h0 : heap}\{\text{pre h0}\} \rightarrow \text{r:(a} * \text{heap)}\{\text{post h0 (fst r) (snd r)}\}$$

Reification expose the monadic *model* of an effect in specification
And only in specification

Relating Reified Stateful Computations

We can now relate our 2 programs with the following lemma :

```
val eq_sum_up_dn (r:ref int) (lo hi:int) (h0:heap) : Lemma
  (requires lo ≤ hi ∧ h0 `contains' r)
  (ensures let _, hup = reify (sum_up r lo hi) h0 in
           let _, hdn = reify (sum_dn r lo hi) h0 in
           hup.[r] == hdn.[r])
```

How the proof goes...

We need an auxilliary lemma relating the two functions :

```
val sum_up_dn_aux (r:ref int) (lo mid hi:int) (h0:heap) : Lemma
  (requires lo ≤ mid ∧ mid ≤ hi ∧ h0 `contains' r)
  (ensures let (_, hup) = reify (sum_up r lo hi) h0 in
           let (_, hmid) = reify (sum_up r lo mid) h0 in
           let (_, hdn) = reify (sum_dn r mid hi) h0 in
           hup.[r] == hmid.[r] + hdn.[r] − h0.[r])
```

How the proof goes...

We need an auxilliary lemma relating the two functions :

```
val sum_up_dn_aux (r:ref int) (lo mid hi:int) (h0:heap) : Lemma
  (requires lo ≤ mid ∧ mid ≤ hi ∧ h0 `contains' r)
  (ensures let (_, hup) = reify (sum_up r lo hi) h0 in
           let (_, hmid) = reify (sum_up r lo mid) h0 in
           let (_, hdn) = reify (sum_dn r mid hi) h0 in
           hup.[r] == hmid.[r] + hdn.[r] − h0.[r])
```

The proof goes by induction on mid :

```
let rec sum_up_dn_aux r hi mid lo h0 =
  if lo ≠ mid then (sum_up_dn_aux r lo (mid−1) hi ; ...)
```

With 2 lemmas not shown here, the SMT fills the rest of the gap

What happens under the hood

Reification is reduced away using the monadic operations

$$\begin{array}{rcl}
\text{reify (return e)} & \rightsquigarrow & \lambda \text{h0} \rightarrow \text{(e, h0)} \\
\text{reify (let x} = \text{e1 in e2)} & \rightsquigarrow & \lambda \text{h0} \rightarrow \text{let x,h1} = \text{e1 h0 in} \\
 & & \text{e2 x h1} \\
\text{reify (get e)} & \rightsquigarrow & \lambda \text{h0} \rightarrow \text{(e, h0)} \\
\text{reify (put e)} & \rightsquigarrow & \lambda \text{h0} \rightarrow \text{((), e)}
\end{array}$$

leaving the SMT to reason only on pure code.

Deriving a program logic for program equivalence

Encoding Nick Benton's (2004) RHL: $c_0 \sim c_1$

```
type command = unit → ST unit

let ( ∼ ) (c0 c1:command) =
    ∀h. let h0, h1 = snd (reify (c0()) h), snd (reify (c1()) h) in
        dom h0 == dom h1 ∧
        ∀(r:ref a{r ∈ h0}). h0.[r] == h1.[r])
```

Deriving a program logic for program equivalence

Encoding Nick Benton's (2004) RHL: $c_0 \sim c_1 : \Phi \Rightarrow \Psi$

```
type command = unit → ST unit

let related (c0 c1:command) (pre post: heap → heap → prop) =
    ∀h0 h1. pre h0 h1 ⟹
        let h0', h1' = snd (reify (c0()) h0), snd (reify (c1()) h1) in
        post h0' h1'
```

Sweeping many details handled in our paper under the rug (notably, termination and equi-termination)

Deriving a program logic for program equivalence

Encoding Nick Benton's (2004) RHL: $c_0 \sim c_1 : \Phi \Rightarrow \Psi$
Prove each of his syntax-directed proof rules as lemmas in F*:

- Relational assignment:

```
val rel_assign post x y e0 e1
  : Lemma (let pre h0 h1 = post (h0.[x] <− e0)
                                (h1.[y] <− e1) in
           related (x := e0) (y := e1) pre post)
```

Deriving a program logic for program equivalence

Encoding Nick Benton's (2004) RHL: $c_0 \sim c_1 : \Phi \Rightarrow \Psi$
Prove each of his syntax-directed proof rules as lemmas in F*:

- Relational assignment:

```
val rel_assign post x y e0 e1
  : Lemma (let pre h0 h1 = post (h0.[x] <− e0)
                                  (h1.[y] <− e1) in
          related (x := e0) (y := e1) pre post)
```

- Relational sequencing:

```
val rel_seq p q r c0 c0' c1 c1'
    : Lemma (related c0 c1 p q ∧ related c0' c1' q r ⟹
             related (c0 ; c0') (c1 ; c1') p r)
```

Mixing Syntax-directed and Semantic Reasoning

- Syntax-directed proof rules for $c_0 \sim c_1 : \Phi \Rightarrow \Psi$ are convenient
- But inherently incomplete, e.g., not possible to prove sum_up $\sim$ sum_dn,
- Where syntax-directed rules don't suffice, fall back on reasoning directly on the reified semantics.

Mixing Syntax-directed and Semantic Reasoning
A Recurring Theme

Hybrid proofs of information-flow security

- Derive a Smith&Volpano-style IFC type system for a embedded imperative language.
  - Proving each rule as a relational lemma on the underlying semantics
- Where the type system is too imprecise, or where programs intentionally declassify information, prove a program-specific non-interference theorem directly.

Several other case studies

- Program equivalence and RHL
- Static information-flow control

Several other case studies

- Program equivalence and RHL
- Static information-flow control
- Security of a dynamic information-flow control monitor

Several other case studies

- Program equivalence and RHL
- Static information-flow control
- Security of a dynamic information-flow control monitor
- Relational characterization of write and read effects (Benton)

Several other case studies

- Program equivalence and RHL
- Static information-flow control
- Security of a dynamic information-flow control monitor
- Relational characterization of write and read effects (Benton)
- Simple game steps of code-based cryptographic proofs (PRHL, FCF, ...)

Several other case studies

- Program equivalence and RHL
- Static information-flow control
- Security of a dynamic information-flow control monitor
- Relational characterization of write and read effects (Benton)
- Simple game steps of code-based cryptographic proofs (PRHL, FCF, ...)
- Algorithmic optimizations
  - McBride's memoization of recursive functions
  - Classic optimizations of imperative Union/Find, via stepwise refinement

Takeaways

- Main idea: Boil down relations on effectful computations to relations on their pure, monadic representations
    - Leverage existing proof assistants capability for reasoning about pure functions
- The relational framework is at the library level, not in the tool
    - Quickly prototype and validate new designs/logics/proof rules
    - No arbitrary restriction on arity of relations
    - Fallback on semantic reasoning when syntactic reasoning is incomplete

Still lots to do ...

- Tactics: to scale and automate syntax directed relational verification
- Non-termination: Only terminating terms can be reified
  - But F* also supports partiality
- Observational purity : going down in the effect lattice

Still lots to do ...
Including applying it at scale for security verification

Project Everest: verify and deploy components in the HTTPS stack

- miTLS Verified reference implementation of TLS
  - Cryptographic game based reduction to ...
  - A classic information flow control argument

Still lots to do ...
Including applying it at scale for security verification

Project Everest: verify and deploy components in the HTTPS stack

- miTLS Verified reference implementation of TLS
  - Cryptographic game based reduction to ...
  - A classic information flow control argument
- HACL* High-Assurance Cryptographic Library
- Vale Verified Assembly Language for Everest
  - Low-level crypto libraries, with proofs of security in the presence of side channels, e.g., timing