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 effectful 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.
Consider proving these two stateful, ML programs equivalent:

```ml
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:

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

let rec sum_down r lo hi =
  if lo \neq 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:

```ml
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:

\[
\text{let rec sum\_up } r \ lo \ hi = \\
\text{ if } \ lo \neq \ hi \ \text{then } (r := r + \ lo; \ \text{sum\_up } r \ (lo+1) \ hi) \\
\]

\[
\text{let rec sum\_down } r \ lo \ hi = \\
\text{ if } \ lo \neq \ hi \ \text{then } (r := r + hi; \ \text{sum\_down } r \ lo \ (hi-1)) \\
\]

Relate the monadic representations of the stateful computations

- Prove \( \text{sum\_up } r \ lo \ hi \sim \text{sum\_dn } r \ lo \ hi \)
- Where \( c_0 \sim c_1 \) relates \( \text{mem} \rightarrow a \ast \text{mem} \) 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 effectful 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 effectful 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 effectful 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*

<table>
<thead>
<tr>
<th>Dependent types</th>
<th>Hoare logic, imperative programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monad-based effects</td>
<td>SMT-based automation</td>
</tr>
<tr>
<td>Coq, Agda</td>
<td>Dafny, Boogie, Vcc</td>
</tr>
<tr>
<td>Lean, Idris</td>
<td>FramaC, Why3, Verifast, ...</td>
</tr>
<tr>
<td>Isabelle (HOL)</td>
<td></td>
</tr>
</tbody>
</table>

But their combination may be.
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_down 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

\[ \Gamma \vdash \{\text{pre}\} \text{code} \{\text{post}\} \]

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

For stateful code:

- **pre**: \( h_0: \text{heap} \rightarrow \text{prop} \)
- **post**: \( h_0: \text{heap} \rightarrow \text{result}: a \rightarrow h_1: \text{heap} \rightarrow \text{prop} \)
Unary Hoare Logic

\[ \Gamma \vdash \{\text{pre}\} \text{code} \{\text{post}\} \]

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

For stateful code:

- \(\triangleright\) \text{pre} : h0:heap \to \text{prop}
- \(\triangleright\) \text{post} : h0:heap \to \text{result:a} \to h1:heap \to \text{prop}

We can intrisically specify our programs:

\textbf{val} sum_up : r:ref int \to lo:int \to hi:int \to \text{ST} \ unit

\((\text{requires} \ \lambda h0 \to lo \leq hi \land h0 \ `\text{contains'} r)\)
\((\text{ensures} \ \lambda h0 () h1 \to h1 \ `\text{contains'} r)\)

\textbf{let rec} sum_up r lo hi =

\text{if} \ lo \neq hi \ \text{then} \ (r := r + lo ; \text{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

\[ \Gamma \vdash e : ST\ a \ (\text{requires } \text{pre}) \ (\text{ensures } \text{post}) \]

\[ \Rightarrow \]

\[ \Gamma \vdash \text{reify } e : \]

\[ \quad h0 : \text{heap}\{\text{pre } h0\} \rightarrow r:(a \ast \text{heap})\{\text{post } h0 \ (\text{fst } r) \ (\text{snd } r)\} \]

Reification expose the monadic \textit{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

\[ \Gamma \vdash e : \text{STATE} \ a \ (\text{requires} \ \text{pre}) \ (\text{ensures} \ \text{post}) \]
\[ \implies \]
\[ \Gamma \vdash \text{reify} \ e : \text{GHOST} \]
\[ h0 : \text{heap}\{\text{pre} \ h0}\rightarrow r:(a \ast \text{heap})\{\text{post} \ h0 \ (\text{fst} \ r) \ (\text{snd} \ r)\} \]

Reification expose the monadic model of an effect in specification
And only in specification
We can now relate our 2 programs with the following lemma:

\[
\begin{align*}
\text{val} & \quad \text{eq\_sum\_up\_dn} \ (r: \text{ref int}) \ (lo \ \text{hi: int}) \ (h0: \text{heap}) : \text{Lemma} \\
& \quad (\text{requires} \ lo \ \leq \ hi \ \land \ h0 \ `\text{contains}' \ r) \\
& \quad (\text{ensures} \ \text{let} \ _, \ hup = \text{reify} \ (\text{sum\_up} \ r \ lo \ hi) \ h0 \ \text{in} \\
& \quad \quad \text{let} \ _, \ hdn = \text{reify} \ (\text{sum\_dn} \ r \ lo \ hi) \ h0 \ \text{in} \\
& \quad \quad hup.[r] == hdn.[r])
\end{align*}
\]
How the proof goes...

We need an auxiliary lemma relating the two functions:

```ocaml
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])
```

With 2 lemmas not shown here, the SMT fills the rest of the gap.
How the proof goes...

We need an auxiliary lemma relating the two functions:

```ocaml
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:

```ocaml
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

\[
\text{reify } \text{(return } e) \rightarrow \lambda h0 \rightarrow (e, h0) \\
\text{reify } \text{(let } x = e1 \text{ in } e2) \rightarrow \lambda h0 \rightarrow \text{let } x, h1 = e1 \text{ h0 in} \\
\hspace{1cm} e2 \times h1 \\
\text{reify } \text{(get } e) \rightarrow \lambda h0 \rightarrow (e, h0) \\
\text{reify } \text{(put } e) \rightarrow \lambda h0 \rightarrow (() , e)
\]

leaving the SMT to reason only on pure code.
Encoding Nick Benton’s (2004) RHL: $c_0 \sim c_1$

```plaintext
type command = unit \rightarrow ST \text{ unit}

let (\sim)(c_0, c_1:command) =
\forall h. let h_0, h_1 = \text{snd (reify (c_0()) h), snd (reify (c_1()) h)} in
  \text{dom h_0} \sqsubseteq \text{dom h_1} \land
  \forall (r: \text{ref a}\{r \in h_0\}). h_0[r] \sqsubseteq h_1[r])
```
Deriving a program logic for program equivalence

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

\[
\text{type command} = \text{unit} \rightarrow \text{ST} \text{ unit}
\]

\[
\text{let related (c0 c1:command) (pre post: heap \rightarrow heap \rightarrow prop)} = \forall h0 h1. \text{pre h0 h1} \implies \text{let h0', h1' = snd (reify (c0()) h0), snd (reify (c1()) h1) in post h0' h1'}
\]

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:

```ocaml
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:

```ocaml
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)
```
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:**

  ```plaintext
  val rel_assign post x y e0 e1 :
  \text{Lemma (let pre h0 h1 = post (h0.[x] \leftarrow e0)
  (h1.[y] \leftarrow e1) in related (x := e0) (y := e1) pre post)}
  ```

- **Relational sequencing:**

  ```plaintext
  val rel_seq p q r c0 c0’ c1 c1’ :
  \text{Lemma (related c0 c1 p q \land related c0’ c1’ q r \implies 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 $\text{sum}_\text{up} \sim \text{sum}_\text{dn}$,
- Where syntax-directed rules don’t suffice, fall back on reasoning directly on the reified semantics.
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, ...)

Algorithmic optimizations
- McBride’s memoization of recursive functions
- Classic optimizations of imperative Union/Find, via stepwise refinement
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