Spy Game – Verifying a Local Generic Solver in Iris

Paulo Emílio de Vilhena, Jacques-Henri Jourdan, François Pottier

January 22, 2020
When is a function constant?

Consider a program “\( f \)” that behaves *extensionally*.

Is it possible to *dynamically* detect that “\( f \)” is a *constant function*?

\[
f: \text{int} \rightarrow \text{int}
\]
When is a function constant?

Consider a program “f” that behaves *extensionally*. Is it possible to *dynamically* detect that “f” is a *constant function*? No. What if “f” is defined on *lazy integers* instead?

```haskell
type lazy_int = unit -> int

f: lazy_int -> int
```
Idea: “If \( f \) does not use its argument, then it must be constant.”

```ocaml
type lazy_int = unit -> int

let is_constant (f: lazy_int -> int) =
    let r = ref true in
    let spy: lazy_int =
        fun () -> r := false; 0
    in
    let _ = f spy in
    !r
```
Idea: “If \( f \) does not use its argument, then it must be constant.”

```ocaml
type lazy_int = unit -> int

let is_constant (f: lazy_int -> int) =
  let r = ref true in
  let spy: lazy_int =
    fun () -> r := false; 0
  in
  let _ = f spy in
  !r
```
Idea: “If $f$ does not use its argument, then it must be constant.”

```ocaml
type lazy_int = unit -> int

let is_constant (f: lazy_int -> int) =
  let r = ref true in
  let spy: lazy_int =
    fun () -> r := false; 0
  in
  let _ = f spy in
  !r
```

• We refer to this programming technique as *spying*.

• Can we verify the correctness of `is_constant`?
Idea: “If $f$ does not use its argument, then it must be constant.”

```ml
type lazy_int = unit -> int

let is_constant (f: lazy_int -> int) =
  let r = ref true in
  let spy: lazy_int =
    fun () -> r := false; 0
  in
  let _ = f spy in
  !r
```
Idea: “If \( f \) does not use its argument, then it must be constant.”

```ocaml
type lazy_int = unit -> int

let is_constant (f: lazy_int -> int) =
  let r = ref true in
  let spy: lazy_int =
    fun () -> r := false; 0
  in
  let _ = f spy in
  !r
```

We refer to this programming technique as spying.

Can we verify the correctness of `is_constant`?
Idea: “If \( f \) does not use its argument, then it must be constant.”

```ocaml
type lazy_int = unit -> int

let is_constant (f: lazy_int -> int) =
  let r = ref true in
  let spy: lazy_int =
    fun () -> r := false; 0
  in
  let _ = f spy in
  !r
```

We refer to this programming technique as **spying**.
Idea: “If \( f \) does not use its argument, then it must be constant.”

```ocaml
type lazy_int = unit -> int

let is_constant (f: lazy_int -> int) =
    let r = ref true in
    let spy: lazy_int =
        fun () -> r := false; 0
    in
    let _ = f spy in
    !r
```

- We refer to this programming technique as **spying**.
- Can we verify the correctness of `is_constant`?
Why is this a relevant question?

- Spying has never been verified in separation logic.
- Spying is used in real-world fixed point computation algorithms.

In the rest of this talk

- Explain and verify spying using Iris – an expressive separation logic.
- By the end, we will have the key ideas to verify is_constant!
- These same ideas allow verifying fixed point computation algorithms.
let is_constant (f: lazy_int -> int) =
    let r = ref true in
    let spy () = r := false; 0 in
    let _ = f spy in !r

The specification is a Hoare triple:

{\textit{f implements } \phi\}  
is_constant f
{b. b = true \Rightarrow \exists c. \forall m. \phi(m) = c}\)

“\textit{x computes m}” is sugar for  \{true\} x () \{y. y = m\}
“\textit{f implements } \phi\)” is sugar for
\forall x, m. \{x \textit{ computes m} \} f x \{y. y = \phi(m)\}
At a first glimpse, the code suggests an intuitive idea:

“If \( r \) contains true, then \( \phi \) is a constant function.”
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements $\phi$ *)
  let r = ref true in
  let spy () =
    r := false; 0
  in
  let _ = f spy in
  !r

At a first glimpse, the code suggests an intuitive idea:

“If $r$ contains true, then $\phi$ is a constant function.”
let is_constant (f: lazy_int -> int) =
(* Assumption: f implements \( \phi \) *)
let r = ref true in
let spy () =
    r := false; 0
in
let _ = f spy in
!r

At a first glimpse, the code suggests an intuitive idea:

"If \( r \) contains true, then \( \phi \) is a constant function."
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements \( \phi \) *)
  let r = ref true in
  let spy () =
    r := false; 0
  in
  let _ = f spy in
  !r

At a first glimpse, the code suggests an intuitive idea:

"If \( r \) contains true, then \( \phi \) is a constant function."
At a first glimpse, the code suggests an intuitive idea:

“If r contains true, then $\phi$ is a constant function.”

The assertion becomes true only after “f spy”. It is not an invariant.
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements $\phi$ *)
let r = ref true in
let spy () =
  r := false; 0
in
let _ = f spy in
!r

A better candidate invariant mentions how many times $f$ calls $\text{spy}$:

“$\#(\text{calls}) = \#(\text{past calls}) + \#(\text{future calls})$

and

if $r$ contains true then $\#(\text{past calls}) = 0$.”

To name the number of future calls, we need prophecy counters.
They are *ghost* code; they do not exist at runtime.

Implemented using Iris’s prophecy variables (Jung et al. 2020).

\[
\begin{align*}
\{\text{true}\} & \quad \text{prophCounter}() \\
\{p. \exists n. p \leadsto n\} & \quad \text{prophDecr } p \\
\{p \leadsto n\} & \quad \text{prophZero } p \\
\{() \cdot 0 < n \cdot p \leadsto (n - 1)\} & \\
\{p \leadsto n\} & \\
\{() \cdot n = 0\}
\end{align*}
\]

**Intuition**

- “The counter predicts how many times it will be decremented.”
let is_constant f =
    let r = ref true in
    let p = prophCounter () in
    let spy () =
        prophDecr p;
        r := false; 0
    in
    let _ = f spy in
    prophZero p;
    !r

The operation `prophCounter ()` yields a natural number \( n \).

Because we use `prophDecr` inside `spy` and `prophZero` at the end, \( n \) is the number of times `spy` will be called!

\[
    n = \#(calls)
\]
let is_constant f =
  let r = ref true in
  let p = prophCounter () in
  let spy () =
    prophDecr p;
    r := false; 0
  in
  let _ = f spy in
  prophZero p;
  !r

Informal:

“\(\#(\text{calls}) = \#(\text{past calls}) + \#(\text{future calls})\) and
if \(r\) contains true then \(\#(\text{past calls}) = 0\).”

Formal:

\[
\text{Inv}(r, p, n) = \exists (k : \text{nat}) (l : \text{nat}) (b : \text{bool}).
\]

\[
p \leadsto l \quad \ast \quad n = k + l \quad \ast \quad r \mapsto b \quad \ast \quad (b = \text{true} \Rightarrow k = 0)
\]
let is_constant f =
  let r = ref true in
  let p = prophCounter () in
  let spy () =
    prophDecr p;
    r := false; 0
  in
  let _ = f spy in
  prophZero p; !r

At the end, by exploiting the invariant, we obtain:

\[ r \mapsto b \quad \ast \quad (b = \text{true} \Rightarrow n = 0) \]

“If \( r \) contains true, then \( \text{spy} \) has never been called.”
let is_constant f =
    let r = ref true in
    let p = prophCounter () in
    let spy () =
        prophDecr p;
        r := false; 0
    in
    let _ = f spy in
    prophZero p; 
    !r

At the end, by exploiting the invariant, we obtain:

\[ r \mapsto b \quad \ast \quad (b = \text{true} \Rightarrow n = 0) \]

"If \( r \) contains \text{true}, then \text{spy} has never been called."

But how to prove that \( \phi \) is constant from there?
Insight 2 – The link between $n$ and $\phi$

```ocaml
let is_constant f = let r = ref true in let p = prophCounter () in let spy () = prophDecr p; r := false; 0 in let _ = f spy in prophZero p; !r
```

“If spy is never called, it can pretend to compute an arbitrary integer.”

\[
\begin{align*}
n = 0 \implies \forall m. \{ \text{true} \} \text{ spy } () \{ y. \ y = m \}
\end{align*}
\]
"If spy is never called, it can pretend to compute an arbitrary integer."

\[ n = 0 \implies \forall m. \text{spy computes } m \]
Insight 2 – The link between $n$ and $\phi$

```
let is_constant f =
  let r = ref true in
let p = prophCounter () in
let spy () =
  prophDecr p; r := false; 0
in
let _ = f spy in
prophZero p; !r
```

“If spy is never called, it can pretend to compute an arbitrary integer.”

$n = 0 \implies \forall m. \text{spy computes } m$

Therefore

$n = 0 \implies \forall m. \left( \{ f \text{ implements } \phi \} \right) \left( \begin{array}{ll}
  \{ f \text{ spy} \} \\
  \{ c. \quad c = \phi(m) \}
\end{array} \right)$
```
let is_constant f = 
  let r = ref true in
let p = prophCounter () in
let spy () =
  prophDecr p; r := false; 0
in
let _ = f spy in
prophZero p; !r
```

“If spy is never called, it can pretend to compute an arbitrary integer.”

\[ n = 0 \implies \forall m. \text{spy computes } m \]

Therefore

\[ n = 0 \implies \forall m. \begin{pmatrix} \{ f \text{ implements } \phi \} \\ f \text{ spy} \\ \{ c. \quad c = \phi(m) \} \end{pmatrix} \]
let is_constant f =
    let r = ref true in
let p = prophCounter () in
let spy () =
    prophDecr p; r := false; 0
in
let _ = f spy in
prophZero p; !r

“If spy is never called, it can pretend to compute an arbitrary integer.”

\[ n = 0 \implies \forall m. \text{spy computes } m \]

Therefore

\[
\begin{align*}
    n = 0 &\implies \\
    \{ f \text{ implements } \phi \} &\implies \\
    \{ c. \forall m. c = \phi(m) \}
\end{align*}
\]
Moving the quantifier is justified by a **restricted conjunction rule**:

\[
\forall x. \{ P \} e \{ y. Q(x, y) \} \quad Q \text{ is pure} \\
\{ P \} e' \{ y. \forall x. Q(x, y) \}
\]

where \( e' \) is a copy of \( e \) instrumented with **prophecy variables**.

The proof in Iris is novel and is yet another use case of prophecies.
Combining the previous steps

```ocaml
let is_constant f =  
    let r = ref true in  
    let p = prophCounter () in  
    let spy () =  
        prophDecr p; r := false; 0  
    in let _ = f spy in  
    prophZero p; !r
```

“If $r$ contains true at the end, then $\text{spy}$ is never called.”

\[
r \mapsto b \quad \ast \quad (b = \text{true} \Rightarrow n = 0)
\]

“If $\text{spy}$ is never called, then $\phi$ is a constant function.”

\[
\begin{align*}
    n = 0 & \quad \Rightarrow \quad \begin{cases}
    \{ f \text{ implements } \phi \} \\
    f \text{ spy} \\
    \{ c. \forall m. c = \phi(m) \}
\end{cases}
\end{align*}
\]

Conclusion: “If $r$ contains true at the end, then $\phi$ is constant.”!
What we have seen so far

- *is_constant* – an example of spying.
- Proof sketch for *is_constant*.
- How prophecy variables are used to handle spying.
- A restricted conjunction rule.

For the rest of the talk

- What is a *local generic solver*.
- Explain the *connection* between spying and local generic solvers.
What is a Local Generic Solver?

A term coined by Fecht and Seidl (1999).

- A *solver* computes the least function “\( \phi \)” that satisfies
  
  \[
  \text{eqs } \phi = \phi
  \]

  where “eqs” is a user-supplied function.

- *Generic* means it is parameterized with a user-defined partial order.

- *Local* means \( \phi \) is computed on demand and need not be defined everywhere.
API of a Local Generic Solver

```ocaml
type valuation = variable -> property
val lfp : (valuation -> valuation) -> valuation

A simple example is to compute Fibonacci:

```ocaml
let rec eqs (phi : valuation) (n : int) = 
  if n <= 1 then 1 else phi (n - 1) + phi (n - 2)

let fib = lfp eqs
```

```
“fib at n depends on fib at n – 1 and n – 2.”

```ml
let eqs (phi : valuation) (n : int) = 
  if n <= 1 then 1 else phi (n - 1) + phi (n - 2)

let fib = lfp eqs
```

- Local generic solvers use dependencies for **efficiency**.
- Dependencies are discovered at **runtime** via spying.
Conclusion

What is in the paper

- Improvements to Iris’s *prophecy variable* API.
- Proof of a *conjunction rule*.
- Use of locks to make our code thread-safe.
- Specification and proof of *modulus*, the general case of spying.
- Specification and proof of a *local generic solver*.

Limitations

- We only prove *partial correctness*.
- We do not prove *deadlock-freedom*. 
Questions?
Spying is subsumed by a single combinator, \textit{modulus}, so named by Longley (1999).

\begin{verbatim}
let modulus ff f =
    let xs = ref [] in
    let spy x =
        xs := x :: !xs;
        f x
    in
    let c = ff spy in
    (c, !xs)
\end{verbatim}

- \texttt{lfp} uses \textit{modulus}.
- \texttt{is\_constant} can be written in terms of \textit{modulus}.
Spying is subsumed by a single combinator, modulus, so named by Longley (1999).

```ocaml
let modulus ff f =  
  let xs = ref [] in  
  let spy x =  
    xs := x :: !xs;  
    f x  
in  
  let c = ff spy in  
(c, !xs)

let is_constant pred =  
  let r = ref true in  
  let spy () =  
    r := false;  
    0  
in  
  let _ = pred spy in  
!r
```

- lfp uses modulus.
- is_constant can be written in terms of modulus.
Spying is subsumed by a single combinator, modulus, so named by Longley (1999).

```ocaml
let modulus ff f =
  let xs = ref [] in
  let spy x =
    xs := x :: !xs;
    f x
  in
  let c = ff spy in
  (c, !xs)
```

```ocaml
let is_constant pred =
  let zero () = 0 in
  match modulus pred zero with
  | _, [] -> true
  | _, _ :: _ -> false
```

- lfp uses modulus.
- is_constant can be written in terms of modulus.
Conjunction rule

\[ \forall x. \{P\} e () \{ y. Q x y\} \quad Q \text{ is pure} \]

\[ \{P\} \text{ withProph } e \{ y. \forall x. Q x y\} \]

where \texttt{withProph } e \texttt{ is the program } e \texttt{ instrumented with prophecies:}

```ocaml
def withProph (e: unit -> 'a) =
    let p = newProph() in
    let y = e () in
    resolveProph p y;
y
```
Prophecy variables

Prophecy Allocation
\{ \text{true} \}

\text{newProph}() \rightarrow \{ p. \ \exists zs. \ p \rightsquigarrow zs \} \subset \mathbb{P}

Prophecy Assignment
\{ p \rightsquigarrow zs \}

\text{resolveProph} p x \rightarrow \{ (). \ \exists zs. \ zs = x :: zs' \cdot p \rightsquigarrow zs' \} \subset \mathbb{P}

Prophecy Disposal
\{ p \rightsquigarrow zs \}

\text{disposeProph} p \rightarrow \{ (). \ zs = [] \} \subset \mathbb{P}

Improvements

- The operation \text{disposeProph} is new.
- The list \text{zs} can have an user-defined type.
\[
\forall \text{eqs } \mathcal{E}. (\mathcal{E} \text{ is monotone}) \implies \{\text{eqs } \textit{implements } \mathcal{E}\} \subseteq \text{lfp eqs} \{\phi.\phi \textit{ implements } \bar{\mu}\mathcal{E}\}
\]

**Remarks**

- Partial correctness: termination is not guaranteed.
- Possible deadlocks depending on the user implementation of \(\mathcal{E}\).
Hofmann et al. (2010a) present a Coq proof of a local generic solver:

- they model the solver as a computation in a state monad,
- and they assume the client can be modeled as a *strategy tree*.

Why it is permitted to model the client in this way is the subject of two separate papers (Hofmann et al. 2010b; Bauer et al. 2013).