Spy Game – Verifying a Local Generic Solver in Iris

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

January 22, 2020

LABORATOIRE DE RECHERCHE EN INFORMATIQUE

Consider a program "`f`" that behaves *extensionally*.

Is it possible to *dynamically* detect that "`f`" is a *constant function*?

```
f: int -> int
```

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?

```
type lazy_int = unit -> int
```

```
f: lazy_int -> int
```

Idea: *"If* `f` *does not use its argument, then it must be constant."*

```
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
```

Approximate solution

Idea: *"If* `f` *does not use its argument, then it must be constant."*

```
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."*

```
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
```

Approximate solution

Idea: *"If* `f` *does* *not* *use its argument, then it must be* *constant."*

```
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
```

Approximate solution

Idea: *"If* `f` *does not use its argument, then it must be constant."*

```
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."*

```
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 ⟵
```

Approximate solution

Idea: *"If* `f` *does not use its argument, then it must be constant."*

```
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.

Specification of is_constant

```
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:

$$\{\texttt{f}\ \textit{\textbf{implements}}\ \phi\}$$
$$\texttt{is\_constant f}$$
$$\{b.\ b = \mathit{true} \Rightarrow \exists c.\ \forall m.\ \phi(m) = c\}$$

"$\texttt{x}$ ***computes*** m" is sugar for $\{\mathit{true}\}\ \texttt{x ()}\ \{y.\ y = m\}$

"$\texttt{f}$ ***implements*** ϕ" is sugar for

$$\forall\, \texttt{x}, m.\ \{\texttt{x}\ \textit{\textbf{computes}}\ m\}\ \texttt{f x}\ \{y.\ y = \phi(m)\}$$

```
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements ϕ *)
  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* ϕ *is a constant function.*"

```
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements ϕ *)
  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* ϕ *is a constant function.*"

```
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements φ *)
  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* ϕ *is a constant function.*"

```
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements ϕ *)
  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* ϕ *is a constant function.*"

```
let is_constant (f: lazy_int -> int) =
  (* Assumption: f implements ϕ *)
  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* ϕ *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 φ *)
  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 `spy`:

"#(calls) = #(past calls) + #(future calls)
and
if `r` *contains* `true` *then #(past calls) = 0."*

To name the number of future calls, we need *prophecy counters*.

Prophecy Counters

They are *ghost* code; they do not exist at runtime.

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

$$\{true\}\ \ prophCounter()\ \ \{p.\ \exists n.\ p \rightsquigarrow n\}$$

$$\{p \rightsquigarrow n\}\ \ prophDecr\ p\ \ \{().\ 0 < n \ * \ p \rightsquigarrow (n-1)\}$$

$$\{p \rightsquigarrow n\}\ \ prophZero\ p\ \ \{().\ n = 0\}$$

Intuition

- *"The counter predicts how many times it will be decremented."*

The Invariant

```
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:

"#(calls) = #(past calls) + #(future calls)
and
if `r` *contains* `true` *then #(past calls) = 0."*

Formal:

$$\mathsf{Inv}(\mathtt{r}, \mathtt{p}, n) = \exists\, (k : \mathit{nat})\, (l : \mathit{nat})\, (b : \mathit{bool}).$$
$$\mathtt{p} \rightsquigarrow l \;\; * \;\; n = k + l \;\; *$$
$$\mathtt{r} \mapsto b \;\; * \;\; (b = \mathtt{true} \Rightarrow k = 0)$$

The Invariant

```
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:

$$\mathtt{r} \mapsto b \;\ast\; (b = \mathtt{true} \Rightarrow n = 0)$$

"If r *contains* true*, then* 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:

$$\mathtt{r} \mapsto b \;\; * \;\; (b = \mathtt{true} \Rightarrow n = 0)$$

"If r *contains* true*, then* spy *has never been called."*

But how to prove that ϕ is constant from there?

Insight 2 – The link between n and ϕ

```
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."*

$$n = 0 \implies \forall m.\ \{true\}\ \texttt{spy}\ \ ()\ \{y.\ y = m\}$$

Insight 2 – The link between n and ϕ

```
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."*

$$n = 0 \implies \forall m.\ \texttt{spy}\ \textbf{\textit{computes}}\ m$$

Insight 2 – The link between n and ϕ

```
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."*

$$n = 0 \implies \forall m.\ \texttt{spy}\ \textbf{\textit{computes}}\ m$$

Therefore

$$n = 0 \implies \forall m. \left(\begin{array}{l} \{\texttt{f}\ \textbf{\textit{implements}}\ \phi\} \\ \qquad\quad \texttt{f spy} \\ \{c. \qquad c = \phi(m)\} \end{array} \right)$$

Insight 2 – The link between n and ϕ

```
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."*

$$n = 0 \implies \forall m.\ \texttt{spy}\ \textbf{\textit{computes}}\ m$$

Therefore

$$n = 0 \implies \forall m. \begin{pmatrix} \{\texttt{f}\ \textbf{\textit{implements}}\ \phi\} \\ \texttt{f spy} \\ \{c.\quad c = \phi(m)\} \end{pmatrix}$$

Insight 2 – The link between n and ϕ

```
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."*

$$n = 0 \implies \forall m.\ \texttt{spy}\ \textbf{\textit{computes}}\ m$$

Therefore

$$n = 0 \implies \left(\begin{array}{c} \{\texttt{f}\ \textbf{\textit{implements}}\ \phi\} \\ \texttt{f spy} \\ \{c.\ \forall m.\ c = \phi(m)\} \end{array} \right)$$

Moving the quantifier is justified by a *restricted conjunction rule:*

$$\frac{\forall x.\ \{P\}\ e\ \{y.\ Q(x,y)\} \qquad Q \text{ is pure}}{\{P\}\ e'\ \{y.\ \forall x. Q(x,y)\}}$$

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

```
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* `spy` *is never called."*

$$\mathtt{r} \mapsto b \ \ * \ \ (b = \mathtt{true} \Rightarrow n = 0)$$

"If `spy` *is never called, then* ϕ *is a constant function."*

$$n = 0 \implies \begin{pmatrix} \{\mathtt{f}\ \textit{\textbf{implements}}\ \phi\} \\ \mathtt{f\ spy} \\ \{c.\, \forall m.\ c = \phi(m)\} \end{pmatrix}$$

Conclusion: *"If* `r` *contains* `true` *at the end, then* ϕ *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

```
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

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

A *simple* example is to compute Fibonacci:

```
type valuation = int -> int
let eqs (phi: valuation) (n: int) =
  if n <= 1 then 1 else phi (n - 1) + phi (n - 2)
in
let fib = lfp eqs
```

"`fib` *at* `n` *depends on* `fib` *at* `n - 1` *and* `n - 2`."

```
type valuation = int -> int
let eqs (phi: valuation) (n: int) =
  if n <= 1 then 1 else phi (n - 1) + phi (n - 2)
in
let fib = lfp eqs
```

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

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, `modulus`, so named by Longley (1999).

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

- `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).

```
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).

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

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

$$\frac{\forall x.\ \{P\}\ \mathtt{e}\ \ ()\ \{y.\ Q\,x\,y\} \qquad Q \text{ is pure}}{\{P\}\ \mathtt{withProph}\ \ \mathtt{e}\ \{y.\ \forall x. Q\,x\,y\}}$$

where `withProph e` is the program `e` instrumented with prophecies:

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

PROPHECY ALLOCATION

$$\{true\}$$
$$newProph()$$
$$\{p.\ \exists zs.\ p \rightsquigarrow zs\}$$

PROPHECY ASSIGNMENT

$$\{p \rightsquigarrow zs\}$$
$$resolveProph\ p\ x$$
$$\{().\ \exists zs'.\ zs = x :: zs' * p \rightsquigarrow zs'\}$$

PROPHECY DISPOSAL

$$\{p \rightsquigarrow zs\}$$
$$disposeProph\ p$$
$$\{().\ zs = []\}$$

Improvements

- The operation *disposeProph* is new.
- The list *zs* can have an user-defined type.

$$\forall\ \texttt{eqs}\ \mathcal{E}.\ (\mathcal{E} \text{ is monotone}) \quad \Rightarrow \quad \begin{array}{c} \{\texttt{eqs}\ \textit{implements}\ \mathcal{E}\} \\ \texttt{lfp eqs} \\ \{\texttt{phi}.\ \texttt{phi}\ \textit{implements}\ \bar{\mu}\mathcal{E}\} \end{array}$$

Remarks

- $\bar{\mu}\mathcal{E}$ is the optimal least fixed point of $\mathcal{E}$.
- Partial correctness: termination is *not* guaranteed.
- Possible deadlocks depending on the user implementation of $\mathcal{E}$.

Related work

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).