Playing spy games in Iris

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

October 28, 2019
Local generic solvers

Spying: implementation and specification of modulus

Spying: verification of modulus

The conjunction rule

Conclusion

Bibliography
Local generic solvers

A family of related algorithms for computing the least solution of a system of recursive equations:

- Fecht and Seidl (1999) coin the term “local generic solver”.
A solver computes the *least fixed point* of a user-supplied monotone second-order function:

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

`lfp eqs` returns a function $\phi$ that purports to be the least fixed point.

We are interested in *on-demand, incremental, memoizing* solvers.

*Nothing is computed* until $\phi$ is applied to a variable $v$. *Minimal work* is then performed: the least fixed point is computed at $v$ and at the variables that $v$ *depends* upon. It is memoized to avoid recomputation. Dependencies are discovered at runtime via *spying.*
F. P. (2009) offers the verification of a local generic solver as a challenge.

Why is it difficult?

A solver offers a pure API, yet uses mutable internal state:

• for memoization – use a lock and its invariant;
F. P. (2009) offers the verification of a local generic solver as a *challenge*.

Why is it difficult?

A solver offers a pure API, yet uses mutable internal state:
- for memoization – use a lock and its invariant;
- for *spying* on the user-supplied function `eqs`. 
Hofmann et al. (2010a) present a Coq proof of a local generic solver, but:

- 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).
What we would like

We would like to obtain a guarantee:

• that concerns an *imperative* solver, not a model of it;
• that holds in the presence of arbitrary *imperative* clients, as long as they respect their end of the specification.

The user-supplied function `eqs` must behave as a pure function, but can have unobservable side effects (state, nondeterminism, concurrency).
In short, we want a *modular* specification in higher-order separation logic:

\[ E \text{ is monotone } \Rightarrow \]
\[ \{ \text{eqs implements flip } E \} \]
\[ \text{lfp eqs} \]
\[ \{ \text{get. get implements } \bar{\mu}E \} \]

\[ \bar{\mu}E \text{ is the optimal least fixed point of } E. \]
The essence of spying can be distilled in a single combinator, *modulus*, so named by Longley (1999).

```plaintext
val modulus : ((('a -> 'b) -> 'c) -> 'a) -> ((('a -> 'b) -> 'c) * ('a list))
```

The call “modulus $\text{ff} \ f$” returns a *pair* of

- the result of the call “$\text{ff} \ f$”, and
- the list of arguments with which $\text{ff}$ has queried $f$ during this call.

This is a complete list of points on which $\text{ff}$ *depends*. 
Here is a simple-minded imperative implementation of modulus:

```plaintext
let modulus ff f =
    let xs = ref [] in
    let spy x =
        (* Record a dependency on x: *)
        xs := x :: !xs;
        (* Forward the call to f: *)
        f x
    in
    let c = ff spy in
    (c, !xs)
```

Longley (1999) gives this code and claims (without proof) that it has the desired denotational semantics in the setting of a pure $\lambda$-calculus.
What is a plausible specification of modulus?

\[\{f \text{ implements } \phi \ast ff \text{ implements } F\}\]

\[\text{modulus } ff \ f\]

\[\{(c, ws). \ [c = F(\phi)]\}\]

The postcondition means that \(c\) is the result of the call “ff ff f”...

“\(f \text{ implements } \phi\)” is sugar for the triple \(\forall x. \ \{true\} \ f \times \{y. \ [y = \phi(x)]\}\).

“\(ff \text{ implements } F\)” means \(\forall f, \phi. \ \{f \text{ implements } \phi\} \ ff \ f \ \{c. \ [c = F(\phi)]\}\).
What is a plausible specification of modulus?

\[
\{ f \text{ implements } \phi \ast \text{ ff implements } \mathcal{F} \}
\]

\[
\text{modulus ff } f
\]

\[
\{(c, ws). \ [\forall \phi'. \ \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi')] \}
\]

The postcondition means that \( c \) is the result of the call “ff ff”... and that \( c \) does not depend on the values taken by \( f \) outside of the list \( ws \).

“\( f \text{ implements } \phi \)” is sugar for the triple \( \forall x. \ {\text{true}} \} f \times \{ y. \ [y = \phi(x)] \} \).

“\( \text{ff implements } \mathcal{F} \)” means \( \forall f, \phi. \ {f \text{ implements } \phi} \ \text{ff } f \ \{c. \ [c = \mathcal{F}(\phi)] \} \).
Local generic solvers

Spying: implementation and specification of modulus

Spying: verification of modulus

The conjunction rule

Conclusion

Bibliography
Why verifying modulus seems challenging

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

\{ f implements \( \phi \star \) ff implements \( \mathcal{F} \}\}

modulus ff f
\{(c, ws). [\forall \phi'. \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi')]\}

\( ff \) expects an \textit{apparently pure} function as an argument, so we \textit{must} prove “\textit{spy implements } \phi'\)” for \textit{some} \( \phi' \), and we will get \( c = \mathcal{F}(\phi') \). However,

- Proving \( c = \mathcal{F}(\phi') \) for \textit{one} function \( \phi' \) is not good enough. It seems as though as we need \textit{spy} to implement \textit{all} functions \( \phi' \) \textit{at once}.
- The set of functions \( \phi' \) over which we would like to quantify is \textit{not known in advance} — it depends on \( ws \), a \textit{result} of modulus.
- What invariant describes \( xs \)? \textit{Only in the end} does it hold a \textit{complete} list \( ws \) of dependencies.
• We need *spy* to implement all functions $\phi'$ at once...

• The list $ws$ is not known in advance...

• What invariant describes $xs$?
Ingredients of a solution

• We need *spy* to implement all functions $\phi'$ at once...
  — Use a *conjunction rule* to focus on one function $\phi'$ at a time.

• The list $ws$ is not known in advance...

• What invariant describes $xs$?
Ingredients of a solution

• We need *spy* to implement all functions \( \phi' \) at once...
  — Use a *conjunction rule* to focus on one function \( \phi' \) at a time.

• The list \( ws \) is not known in advance...
  — Use a *prophecy variable* to name this list ahead of time.

• What invariant describes \( xs \)?
Ingredients of a solution

- We need *spy* to implement all functions \( \phi' \) at once...
  - Use a *conjunction rule* to focus on one function \( \phi' \) at a time.

- The list \( ws \) is not known in advance...
  - Use a *prophecy variable* to name this list ahead of time.

- What invariant describes \( xs \)?
  - The elements *currently recorded* in \( !xs \), concatenated with those that *will be recorded* in the future, form the list \( ws \).
Instead of establishing this *strong* specification for modulus...

\[
\begin{align*}
&\{ f \text{ implements } \phi \ast ff \text{ implements } \mathcal{F} \} \\
&\text{modulus } ff f
\end{align*}
\]

\[
\begin{align*}
&\{ (c, ws). \forall \phi'. \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \}
\end{align*}
\]
A weaker specification for modulus

\[ \forall \phi'. \left( \begin{array}{l}
\{ f \text{ implements } \phi' \} \\
\text{modulus } ff f \\
\{ (c, ws). \ [ \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') ] \} 
\end{array} \right) \]

...let us first establish a \textit{weaker} specification.

Then (later), use an infinitary \textit{conjunction rule} to argue (roughly) that the weaker spec implies the stronger one.
Assume φ′ is given.

```ocaml
let modulus ff f =
    let xs, p, lk = ref [], newProph(), newLock() in
    let spy x =
        let y = f x in
        withLock lk (fun () ->
            xs := x :: !xs; resolveProph p x);
        y
    in
    let c = ff spy in
    acquireLock lk; disposeProph p; (c, !xs)
```

Step 1. Allocate a prophecy variable p.
Introduce the name ws to stand for the list of future writes to p.
Proof of modulus

Assume $\phi'$ is given.

```ocaml
let modulus ff f =
  let xs, p, lk = ref [], newProph(), newLock() in
  let spy x =
    let y = f x in
    withLock lk (fun () ->
      xs := x :: !xs; resolveProph p x);
    y
  in
  let c = ff spy in
  acquireLock lk; disposeProph p; (c, !xs)
```

Step 2. Allocate a lock $lk$, which owns $xs$ and $p$. Its invariant is that the list $ws$ of *all writes* to $p$ can be split into two parts:

- the *past writes*, the reverse of the current contents of $xs$;
- the remaining *future writes* to $p$.
Assume $\phi'$ is given.

```ocaml
let modulus ff f =
  let xs, p, lk = ref [], newProph(), newLock() in
  let spy x =
    let y = f x in
    withLock lk (fun () ->
      xs := x :: !xs; resolveProph p x);
    y
  in
  let c = ff spy in
  acquireLock lk; disposeProph p; (c, !xs)
```

Step 2. Allocate a lock $lk$, which owns $xs$ and $p$. Its invariant is that the list $ws$ of all writes to $p$ can be split into two parts:

- the past writes, the reverse of the current contents of $xs$;
- the remaining future writes to $p$.

Moving $x$ from one part to the other preserves the invariant.
Assume $\phi'$ is given.

```ocaml
let modulus ff f =
  let xs, p, lk = ref [], newProph(), newLock() in
  let spy x =
    let y = f x in
    withLock lk (fun () ->
      xs := x :: !xs; resolveProph p x);
    y
  in
  let c = ff spy in
  acquireLock lk; disposeProph p; (c, !xs)
```

Because `acquireLock` exhales the invariant and `disposeProph` guarantees there are no more future writes, !xs on the last line yields $ws$ (reversed).

Thus, the name $ws$ in the postcondition of `modulus` and the name $ws$ introduced by `newProph` denote `the same set` of points.
Proof of modulus

Assume $\phi'$ is given.

```ocaml
let modulus ff f =
  let xs, p, lk = ref [], newProph(), newLock() in
  let spy x =
    let y = f x in
    withLock lk (fun () ->
      xs := x :: !xs; resolveProph p x);
    y
  in
  let c = ff spy in
  acquireLock lk; disposeProph p; (c, !xs)
```

Step 3. Reason by cases:
- If $\phi' \neq_{ws} \phi$ does not hold, then the postcondition of modulus is true. Then, it suffices to prove that modulus is safe, which is not difficult.
- If $\phi' =_{ws} \phi$ does hold, continue on to the next slides...
Proof of modulus

Assume $\phi'$ is given. Assume $\phi' =_{ws} \phi$ holds.

```ocaml
let modulus ff f =
  let xs, p, lk = ref [], newProph(), newLock() in
  let spy x =
    let y = f x in
    withLock lk (fun () ->
       xs := x :: !xs; resolveProph p x);
    y
  in
  let c = ff spy in
  acquireLock lk; disposeProph p; (c, !xs)
```

Step 4. Prove that $spy$ implements $\phi'$.

- We have $y = \phi(x)$. We wish to prove $y = \phi'(x)$. 

---
Assume $\phi'$ is given. Assume $\phi' =_{ws} \phi$ holds.

```ocaml
let modulus ff f =
  let xs, p, lk = ref [], newProph(), newLock() in
  let spy x =
    let y = f x in
    withLock lk (fun () ->
      xs := x :: !xs; resolveProph p x);
    y
  in
  let c = ff spy in
  acquireLock lk; disposeProph p; (c, !xs)
```

Step 4. Prove that $spy$ implements $\phi'$.

- We have $y = \phi(x)$. We wish to prove $y = \phi'(x)$.
- Because $\phi$ and $\phi'$ coincide on $ws$, the goal boils down to $x \in ws$. 
Assume $\phi'$ is given. Assume $\phi' =_{ws} \phi$ holds.

```ocaml
let modulus ff f =
  let xs, p, lk = ref [], newProph(), newLock() in
  let spy x =
    let y = f x in
    withLock lk (fun () ->
      xs := x :: !xs; resolveProph p x);
    y
  in
  let c = ff spy in
  acquireLock lk; disposeProph p; (c, !xs)
```

Step 4. Prove that $spy$ implements $\phi'$.

- We have $y = \phi(x)$. We wish to prove $y = \phi'(x)$.
- Because $\phi$ and $\phi'$ coincide on $ws$, the goal boils down to $x \in ws$.
- $x \in ws$ holds because we make it hold by writing $x$ to $p$.
  — “there, let me bend reality for you”
Proof of modulus

Assume $\phi'$ is given. Assume $\phi' =_{ws} \phi$ holds.

```ocaml
let modulus ff f =
    let xs, p, lk = ref [], newProph(), newLock() in
    let spy x =
        let y = f x in
        withLock lk (fun () ->
            xs := x :: !xs; resolveProph p x);
        y
    in
    let c = ff spy in
    acquireLock lk; disposeProph p; (c, !xs)
```

Step 5. From “$ff$ implements $\mathcal{F}$” and “$spy$ implements $\phi'$”, deduce that
the call “$ff$ spy” is permitted and that $c = \mathcal{F}(\phi')$ holds.

$c = \mathcal{F}(\phi')$ is the postcondition of $modulus$. We are done!
Recall that, from this *weak* specification of *modulus*...

\[\forall \phi'. \left( \begin{array}{l}
\{ f \text{ implements } \phi \ast ff \text{ implements } \mathcal{F} \} \\
\text{modulus ff f} \\
\{(c, ws). \left[ \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \right]\} \end{array} \right) \]
Recall that, from this weak specification of modulus...

$$\forall \phi'.\ {\phi'} =_{ws} \phi \implies c = F(\phi')$$

...we need to deduce this stronger specification.

This is where an infinitary conjunction rule is needed.
An array of conjunction rules

**Binary, Non-Dependent**

\[
\{ P \} \ e \ \{ \_ \ [ Q_1 ] \} \\
\{ P \} \ e \ \{ \_ \ [ Q_2 ] \} \\
\{ P \} \ e \ \{ \_ \ [ Q_1 \land Q_2 ] \}
\]

**Binary, Dependent**

\[
\{ P \} \ e \ \{ y \ [ Q_1 \ y ] \} \\
\{ P \} \ e \ \{ y \ [ Q_2 \ y ] \} \\
\{ P \} \ e \ \{ y \ [ Q_1 \ y \land Q_2 \ y ] \}
\]

**Infinitary, Non-Dependent**

\[
\forall x. \ \{ P \} \ e \ \{ \_ \ [ Q \ x ] \} \\
\{ P \} \ e \ \{ \_ \ [ \forall x. Q \ x ] \}
\]

**Infinitary, Dependent**

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

The non-dependent variants are *sound*.

The dependent variants may be sound (*open question!*).
We can derive an approximation that’s good enough for our purposes.
All of the previous rules are restricted to *pure* postconditions.

An unrestricted conjunction rule is *unsound* in the presence of ghost state.

\[
\text{IMPURE (UN Sound!)}
\]

\[
\begin{align*}
\{P\} & \text{ e } \{\_ \cdot Q_1\} \\
\{P\} & \text{ e } \{\_ \cdot Q_2\} \\
\hline
\{P\} & \text{ e } \{\_ \cdot Q_1 \land Q_2\}
\end{align*}
\]

*Open question!*

Would this rule be sound if every ghost update was apparent in the code?
Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x. \{P\} e \{\_ \_ \_ \_ \_ \_ [Qx]\}$
Goal: $\{P\} e \{\_ \_ \_ \_ \_ \_ \_ [\forall x. Qx]\}$

\[
\{P\}
\]
Proof outline — infinitary, non-dependent case

**Hypothesis:** \( \forall x. \{P\} e \{\_ \_ \_ [Q \_ x]\} \)

**Goal:** \( \{P\} e \{\_ \_ \_ [\forall x. Q \_ x]\} \)

Case split: \( (\forall x. Q \_ x) \lor (\exists x. \neg Q \_ x) \)
Proof outline — infinitary, non-dependent case

Hypothesis: \( \forall x. \{ P \} e \{ \neg \\forall x. Q x \} \)

Goal: \( \{ P \} e \{ \forall x. Q x \} \)

Case split: \((\forall x. Q x) \lor (\exists x. \neg Q x)\)

\[ \{ P \} \]

\[ \{ P \ast \forall x. Q x \} \]

\[ e \]

\[ \{ \forall x. Q x \} \]

\( \{ P \ast \forall x. Q x \} e \{ \forall x. Q x \} \)
Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x. \{P\} \ e \ \{\_ \ [Q \ x]\}$
Goal: $\{P\} \ e \ \{\_ \ [\forall x. \ Q \ x]\}$

Case split: $(\forall x. \ Q \ x) \ \lor \ (\exists x. \ \neg Q \ x)$

$\{P \ \ast [\forall x. \ Q \ x]\} \ e \ \{[\forall x. \ Q \ x]\}$

$\{P \ \ast [\exists x. \ \neg Q \ x]\}$
Proof outline — infinitary, non-dependent case

**Hypothesis:** \( \forall x. \{ P \} e \{ \neg. \lceil Q x \rceil \} \)

**Goal:** \( \{ P \} e \{ \neg. \lceil \forall x. Q x \rceil \} \)

---

\( \{ P \} \)

Case split: \( (\forall x. Q x) \lor (\exists x. \neg Q x) \)

\( \{ P \ast \lceil \forall x. Q x \rceil \} \)

\( e \)

\( \{ \lceil \forall x. Q x \rceil \} \)

\( \{ P \ast \lceil \exists x. \neg Q x \rceil \} \)

\( e \)

\( \{ \exists x. \lceil Q x \rceil \ast \lceil \neg Q x \rceil \} \)
Proof outline — infinitary, non-dependent case

**Hypothesis:** \( \forall x. \{ P \} e \{ \_ \_ \_ [ Q x ] \} \)

**Goal:** \( \{ P \} e \{ \_ \_ \_ [ \forall x. Q x ] \} \)

Case split:

\[
(P)
\]

\[
\exists x. \neg Q x
\]

\[
\forall x. Q x
\]

\[
\exists x. \neg Q x
\]

\[
false
\]
Proof outline — infinitary, non-dependent case

Hypothesis: \( \forall x. \{ P \} e \{ \neg \[ Q x \] \} \)

Goal: \( \{ P \} e \{ \neg \[ \forall x. \ Q x \] \} \)

Case split: \((\forall x. \ Q x) \lor (\exists x. \neg \ Q x)\)

\[
\begin{align*}
\{ P \} \\
\{ P * \[ \forall x. \ Q x \] \} & e \{ [\forall x. \ Q x] \} \\
\{ P * \[ \exists x. \neg \ Q x \] \} & e \{ \exists x. \ [ Q x ] * [\neg \ Q x] \} \\
& \{ \exists x. \ [ Q x ] * [\neg \ Q x] \} \\
& \{ false \} \\
\{ [\forall x. \ Q x] \} & \\
\end{align*}
\]
Same idea, but a *prophecy variable* must be used to name $y$ ahead of time and allow the case split $(\forall x. Q x y) \lor \neg(\forall x. Q x y)$.

**Infinitary, Dependent**

\[
\forall x. \{P\} \ e \ \{y. \lceil Q x y \rceil\} \\
\{P\} \ e' \ \{y. \lceil \forall x. Q x y \rceil\}
\]

Because of this, $e'$ in the conclusion is a copy of $e$ instrumented with \texttt{newProph} and \texttt{resolveProph} instructions. (Ouch.)
• Extension of Iris’s prophecy API: `disposeProph`; typed prophecies.
• Proof of the conjunction rule.
• Specification and proof of `modulus`.
• Specification and proof of a slightly simplified version of `Fix`:

\[
\mathcal{E} \text{ is monotone } \Rightarrow \\
\{ \text{eqs implements } \text{flip } \mathcal{E} \} \\
\text{lfp eqs} \\
\{ \text{get. get implements } \tilde{\mu} \mathcal{E} \}
\]

where \( \tilde{\mu} \mathcal{E} \) is the optimal least fixed point of \( \mathcal{E} \).
A few optimizations are missing, e.g.,

- **Fix** uses a more efficient representation of the dependency graph.

Caveats:

- Termination is not proved.
- Deadlock-freedom is not proved.

Wishes:

- Is there any way of *not* polluting the code with operations on prophecy variables?
Spying is another archetypical use of hidden state.

Prophecy variables are fun, and they can be useful not just in concurrent code, but also in sequential code.
Local generic solvers

Spying: implementation and specification of modulus

Spying: verification of modulus

The conjunction rule

Conclusion

Bibliography
References I


