Playing spy games in Iris

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

November 11, 2019
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:

```haskell
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`. 
In short, we want a *modular* specification in higher-order separation logic:

\[ \mathcal{E} \text{ is monotone } \Rightarrow \{ \text{eqs implements flip } \mathcal{E} \} \]

\[ \text{lfp eqs} \]

\[ \{ \text{get. get implements } \bar{\mu} \mathcal{E} \} \]

\( \bar{\mu} \mathcal{E} \) is the optimal least fixed point of \( \mathcal{E} \).
Local generic solvers

Spying: implementation and specification of modulus

Spying: verification of modulus

The conjunction rule

Conclusion

Bibliography
The essence of spying can be distilled in a single combinator, modulus, so named by Longley (1999).

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

The call “modulus $f\,f\,f$” returns a pair of

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

This is a complete list of points on which $f\,f$ depends.
Here is a simple-minded imperative implementation of modulus:

```ocaml
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 λ-calculus.
What is a plausible specification of modulus?

\[
\begin{align*}
\{ f \text{ implements } \phi \ast \text{ff implements } \mathcal{F} \} \\
\text{modulus } \text{ff } f \\
\{ (c, ws). \left[ c = \mathcal{F}(\phi) \right] \}
\end{align*}
\]

The postcondition means that \( c \) is the result of the call \( \text{ff } f \)...

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

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

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

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

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

The postcondition means that \( c \) is the result of the call “\( ff \ f \)” ... 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 \( \forall x . \{ \text{true} \} f \times \{ y . [ y = \phi(x) ] \} \). “\( ff \text{ implements } F \)” means \( \forall f, \phi . \{ f \text{ implements } \phi \} ff f \{ c. [ c = F(\phi) ] \} \).
Local generic solvers

Spying: implementation and specification of modulus

Spying: verification of modulus

The conjunction rule

Conclusion

Bibliography
Why verifying modulus seems challenging

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

```latex
\{ f \text{ implements } \phi \} \quad \{ \text{modulus } ff f \}
\{(c, ws). [\forall \phi'. \phi' =_{ws} \phi \Rightarrow c = F(\phi')]\}
```

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

- Proving \(c = F(\phi')\) for 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.
Ingredients of a solution

• We need \texttt{spy} to implement all functions \( \phi' \) at once...

• The list \( \texttt{ws} \) is not known in advance...

• What invariant describes \( \texttt{xs} \)?
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 $\mathit{ws}$ is not known in advance...
  — Use a *prophecy variable* to name this list ahead of time.

• What invariant describes $\mathit{xs}$?
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 $\text{ws}$ is not known in advance...
  - Use a *prophecy variable* to name this list ahead of time.

- What invariant describes $\text{xs}$?
  - The elements *currently recorded* in $\text{!xs}$, concatenated with those that *will be recorded* in the future, form the list $\text{ws}$. 
In Hoare Logic and Separation Logic, assertions describe the *current* state.
- e.g., “at this point, \(!xs\) is the empty list \([\,]\)”

The current state, possibly enriched with *ghost state*, reflects the *past*.
There is no way of talking about the *future!*
In Hoare Logic and Separation Logic, assertions describe the current state.

- e.g., “at this point, \( \texttt{!xs} \) is the empty list \([\texttt{[]}\)"

The current state, possibly enriched with ghost state, reflects the past. There is no way of talking about the future!

Enter prophecy variables (Abadi and Lamport 1988; Jung et al. 2020).
A prophecy variable primer

A **ghost variable** with three operations: allocation, assignment, disposal.

The reasoning rules allow referring to the sequence $xs$ of *future writes*.

**Prophecy Allocation**

\[
\text{Prophecy Allocation} \quad \{\text{true}\} \\
\text{newProph}() \\
\{ p. \exists xs. p \text{ will receive } xs \}
\]

**Prophecy Assignment**

\[
\text{Prophecy Assignment} \quad \{ p \text{ will receive } xs \} \\
\text{resolveProph} p x \\
\{ ((). \exists xs'. [xs = x :: xs'] p \text{ will receive } xs' \}
\]

**Prophecy Disposal**

\[
\text{Prophecy Disposal} \quad \{ p \text{ will receive } xs \} \\
\text{disposeProph} p \\
\{((). [xs = []] \}
\]
A weaker specification for modulus

Instead of establishing this \textit{strong} specification for modulus...

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

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

...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.
Proof of modulus

Assume $\phi'$ is given.

```plaintext
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
define modulus 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 $\text{modulus}$ is true. Then, it suffices to prove that $\text{modulus}$ is safe, which is not difficult.

- If $\phi' =_{ws} \phi$ does hold, continue on to the next slides...
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 $\text{spy implements } \phi'$.
- We have $y = \phi(x)$. We wish to prove $y = \phi'(x)$.
Assume $\phi'$ is given. Assume $\phi' =_{ws} \phi$ holds.

Let modulus $ff$ be defined as:

```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 $x \in ws$. 

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 \textit{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 \textit{because we make it hold} by writing $x$ to $p$.
  — “there, let me bend reality for you”
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!
Local generic solvers

Spying: implementation and specification of modulus

Spying: verification of modulus

The conjunction rule

Conclusion

Bibliography
Recall that, from this \textit{weak} specification of \textit{modulus}...

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

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

...we need to deduce this stronger specification.

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

<table>
<thead>
<tr>
<th>Binary, Non-Dependent</th>
<th>Binary, Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>{P} e {_., [Q_1]}</td>
<td>{P} e {y., [Q_1,y]}</td>
</tr>
<tr>
<td>{P} e {_., [Q_2]}</td>
<td>{P} e {y., [Q_2,y]}</td>
</tr>
<tr>
<td>{P} e {_., [Q_1 \land Q_2]}</td>
<td>{P} e {y., [Q_1,y \land Q_2,y]}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Infinitary, Non-Dependent</th>
<th>Infinitary, Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\forall x. {P} e {_., [Q,x]})</td>
<td>(\forall x. {P} e {y., [Q,x,y]})</td>
</tr>
<tr>
<td>{P} e {_., [\forall x. Q,x]}</td>
<td>{P} e {y., [\forall x. Q,x,y]}</td>
</tr>
</tbody>
</table>

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.
An unsound conjunction rule

All of the previous rules are restricted to pure postconditions.

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

\[
\text{Impure (Unsound!)}
\begin{align*}
\{P\} & \text{ e } \{. Q_1\} \\
\{P\} & \text{ e } \{. Q_2\} \\
\hline
\{P\} & \text{ e } \{. Q_1 \land Q_2\}
\end{align*}
\]

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

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

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

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

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

\[
\begin{align*}
\{P\} \\
\text{Case split:} \quad & (\forall x. \ Q \ x) \quad \lor \quad (\exists x. \neg Q \ x) \\
\end{align*}
\]
Proof outline — infinitary, non-dependent case

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

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

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

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

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

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

\[
\begin{align*}
\{ P \} \\
\{ P \ast \ [ \forall x. \ Q \ x ] \} \\
\{ [ \forall x. \ Q \ x ] \} \\
\text{e} \\
\{ [ \forall x. \ Q \ x ] \}
\end{align*}
\]

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

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

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

---

\[
\begin{align*}
\{ P \} \\
\text{Case split:} & \quad (\forall x. Q x) \lor (\exists x. \neg Q x) \\
\{ P * [\forall x. Q x] \} & \quad e \quad \{ [\forall x. Q x] \} \\
\{ P * [\exists x. \neg Q x] \} & \quad e \quad \{ \exists x. P * [\neg Q x] \} \\
\{ \exists x. [Q x] * [\neg Q x] \} & \quad e \quad \{ \exists x. [Q x] * [\neg Q x] \}
\end{align*}
\]
Proof outline — infinitary, non-dependent case

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

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

---

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

\[
\begin{align*}
\{ P \} & \\
\{ P * [\forall x. Q x] \} e & \\
\{ [\forall x. Q x] \} \\
\{ P * [\exists x. \neg Q x] \} e & \\
\{ \exists x. P * [\neg Q x] \} & \\
\{ \exists x. [Q x] * [\neg Q x] \} & \\
\{ false \}
\end{align*}
\]
Proof outline — infinitary, non-dependent case

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

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

---

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

\[
\begin{align*}
\{P\} & \quad \{P \ast \lceil \forall x. \ Q \ x \rceil\} \\
& \quad \{\exists x. \ P \ast \lceil \neg \ Q \ x \rceil\} \\
& \quad \{\exists x. \lceil Q \ x \rceil \ast \lceil \neg \ Q \ x \rceil\} \\
& \quad \{false\} \\
& \quad \{\lceil \forall x. \ Q \ x \rceil\}
\end{align*}
\]
The infinitary, dependent case

Same idea, but a *prophecy variable* must be used to name $y$ ahead of time and allow the case split $(\forall x. Q \times y) \lor \lnot (\forall x. Q \times y)$.

\[
\text{INFINITARY, DEPENDENT} \\
\forall x. \{P\} \ e \ \{y. [Q \times y]\} \\
\{P\} \ e' \ \{y. [\forall x.Q \times y]\}
\]

Because of this, $e'$ in the conclusion is a copy of $e$ instrumented with \textit{newProph} and \textit{resolveProph} instructions. (Ouch.)
Contributions

- 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`:

\[
\begin{align*}
\mathcal{E} \text{ is monotone} & \Rightarrow \\
\{ \text{eqs implements } \text{flip } \mathcal{E} \} & \Rightarrow \\
\text{lfp eqs} & \\
\{ \text{get. get implements } \bar{\mu} \mathcal{E} \} & \Rightarrow
\end{align*}
\]

where \( \bar{\mu} \mathcal{E} \) 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*. 


