Playing spy games in Iris

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

October 28, 2019

Contents

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

- Le Charlier and Van Hentenryck (1992).
- Vergauwen and Lewi (1994).
- Fecht and Seidl (1999) coin the term "local generic solver".
- F. P. (2009) releases Fix and asks how to *verify* it.

API of a 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*.

A challenge

F. P. (2009) offers the verification of a local generic solver as a *challenge*.

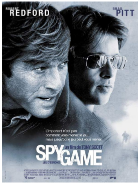
Why is it difficult?

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`.

A partial answer

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).

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).

What we would like

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

$$\begin{gathered}\mathcal{E} \text{ is monotone } \Rightarrow \\ \{\mathit{eqs\ implements\ flip}\,\mathcal{E}\} \\ \mathit{lfp\ eqs} \\ \{\mathit{get}.\ \mathit{get\ implements}\ \bar{\mu}\mathcal{E}\}\end{gathered}$$

$\bar{\mu}\mathcal{E}$ is *the optimal least fixed point of* $\mathcal{E}$.

Contents

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 ff f`" returns a *pair* of

- the result of the call "`ff f`", and
- the list of arguments with which `ff` has queried `f` during this call.

This is a complete list of points on which `ff` *depends*.

Here is a simple-minded imperative implementation of `modulus`:

```
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.

Specification of modulus

What is a plausible specification of `modulus`?

$$\{f \ \mathit{implements}\ \phi * \mathit{ff}\ \mathit{implements}\ \mathcal{F}\}$$
$$\mathit{modulus}\ \mathit{ff}\ f$$
$$\{(c, ws).\ \lceil c = \mathcal{F}(\phi) \rceil\}$$

The postcondition means that c is the result of the call "`ff f`"...

"f *implements* ϕ" is sugar for the triple $\forall x.\ \{\mathit{true}\}\ f\ x\ \{y.\ \lceil y = \phi(x) \rceil\}$.

"*ff implements* $\mathcal{F}$" means $\forall f, \phi.\ \{f\ \mathit{implements}\ \phi\}\ \mathit{ff}\ f\ \{c.\ \lceil c = \mathcal{F}(\phi) \rceil\}$.

What is a plausible specification of `modulus`?

$$\{f \textit{ implements } \phi * \mathit{ff} \textit{ implements } \mathcal{F}\}$$
$$\textit{modulus } \mathit{ff} \ f$$
$$\{(c, ws).\ \lceil \forall \phi'.\ \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \rceil\}$$

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 *implements* ϕ" is sugar for the triple $\forall x.\ \{\mathit{true}\}\ f\ x\ \{y.\ \lceil y = \phi(x) \rceil\}$.

"ff *implements* $\mathcal{F}$" means $\forall f, \phi.\ \{f \textit{ implements } \phi\}\ \mathit{ff}\ f\ \{c.\ \lceil c = \mathcal{F}(\phi) \rceil\}$.

Contents

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)
```

$$\{f\ \mathit{implements}\ \phi * \mathit{ff}\ \mathit{implements}\ \mathcal{F}\}$$
$$\mathit{modulus}\ \mathit{ff}\ f$$
$$\{(c, ws).\ \lceil \forall \phi'.\ \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \rceil\}$$

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

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

Ingredients of a solution

- We need *spy* to implement all functions ϕ' 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 ϕ' at once...
  — Use a *conjunction rule* to focus on one function ϕ' 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 ϕ' at once...
  — Use a *conjunction rule* to focus on one function ϕ' 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 ϕ' at once...
  — Use a *conjunction rule* to focus on one function ϕ' 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`...

$$\left(\begin{array}{c} \{f \ \mathit{implements}\ \phi * \mathit{ff}\ \mathit{implements}\ \mathcal{F}\} \\ \mathit{modulus}\ \mathit{ff}\ f \\ \{(c, ws).\ \lceil \forall \phi'.\ \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \rceil\} \end{array}\right)$$

A weaker specification for modulus

$$\forall \phi'. \left(\begin{array}{c} \{f \ \mathit{implements}\ \phi * \mathit{ff}\ \mathit{implements}\ \mathcal{F}\} \\ \mathit{modulus}\ \mathit{ff}\ f \\ \{(c, ws).\ \lceil \quad \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \rceil\} \end{array} \right)$$

...let us first establish a *weaker* specification.

Then (later), use an infinitary *conjunction rule* to argue (roughly) that the weaker spec implies the stronger one.

Assume ϕ' is given.

```
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*.

Assume ϕ' is given.

```
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 ϕ' is given.

```
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.

Proof of modulus

Assume ϕ' is given.

```
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 ϕ' is given.

```
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 $\phi' =_{ws} \phi$ 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...

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

```
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* ϕ'.

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

Proof of modulus

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

```
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* ϕ'.

- We have $y = \phi(x)$. We wish to prove $y = \phi'(x)$.
- Because ϕ and ϕ' coincide on ws, the goal boils down to $x \in ws$.

Proof of modulus

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

```
  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* ϕ'.

- We have $y = \phi(x)$. We wish to prove $y = \phi'(x)$.
- Because ϕ and ϕ' 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 ϕ' is given. Assume $\phi' =_{ws} \phi$ holds.

```
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* ϕ'", 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!

Contents

Recall that, from this *weak* specification of *modulus*...

$$\forall \phi'. \left(\begin{array}{c} \{f \ \mathit{implements} \ \phi * \mathit{ff} \ \mathit{implements} \ \mathcal{F}\} \\ \mathit{modulus} \ \mathit{ff} \ f \\ \{(c, ws). \lceil \qquad \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \rceil \} \end{array} \right)$$

$$\left(\begin{array}{c} \{f \textit{ implements } \phi * \textit{ff implements } \mathcal{F}\} \\ \textit{modulus ff f} \\ \{(c, ws).\ \lceil \forall \phi'.\ \phi' =_{ws} \phi \Rightarrow c = \mathcal{F}(\phi') \rceil\} \end{array} \right)$$

...we need to deduce this *stronger* specification.

This is where an infinitary *conjunction rule* is needed.

Binary, Non-Dependent

$$\frac{\{P\}\ e\ \{\_.\ \lceil Q_1 \rceil\} \quad \{P\}\ e\ \{\_.\ \lceil Q_2 \rceil\}}{\{P\}\ e\ \{\_.\ \lceil Q_1 \wedge Q_2 \rceil\}}$$

Binary, Dependent

$$\frac{\{P\}\ e\ \{y.\ \lceil Q_1\, y \rceil\} \quad \{P\}\ e\ \{y.\ \lceil Q_2\, y \rceil\}}{\{P\}\ e\ \{y.\ \lceil Q_1\, y \wedge Q_2\, y \rceil\}}$$

Infinitary, Non-Dependent

$$\frac{\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\, x \rceil\}}{\{P\}\ e\ \{\_.\ \lceil \forall x. Q\, x \rceil\}}$$

Infinitary, Dependent

$$\frac{\forall x.\ \{P\}\ e\ \{y.\ \lceil Q\, x\, y \rceil\}}{\{P\}\ e\ \{y.\ \lceil \forall x. Q\, x\, y \rceil\}}$$

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.

$$\textsc{Impure (Unsound!)}\ \frac{\{P\}\ e\ \{\_.\ Q_1\} \qquad \{P\}\ e\ \{\_.\ Q_2\}}{\{P\}\ e\ \{\_.\ Q_1 \land Q_2\}}$$

Open question!

Would this rule be sound if every ghost update was apparent in the code?

Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\, x \rceil\}$
Goal: $\{P\}\ e\ \{\_.\ \lceil \forall x.\ Q\, x \rceil\}$

$$\{P\}$$

Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\,x \rceil\}$
Goal: $\{P\}\ e\ \{\_.\ \lceil \forall x.\ Q\,x \rceil\}$

$$\{P\}$$

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

Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\,x \rceil\}$
Goal: $\{P\}\ e\ \{\_.\ \lceil \forall x.\ Q\,x \rceil\}$

$$\{P\}$$

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

$$\{P * \lceil \forall x.\ Q\ x \rceil\}$$
$$e$$
$$\{\lceil \forall x.\ Q\ x \rceil\}$$

Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\,x \rceil\}$
Goal: $\{P\}\ e\ \{\_.\ \lceil \forall x.\ Q\,x \rceil\}$

$$\{P\}$$

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

	$\{P * \lceil \exists x.\ \neg\ Q\ x \rceil\}$
$\{P * \lceil \forall x.\ Q\ x \rceil\}$ e $\{\lceil \forall x.\ Q\ x \rceil\}$	

Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\,x \rceil\}$
Goal: $\{P\}\ e\ \{\_.\ \lceil \forall x.\ Q\,x \rceil\}$

$$\{P\}$$

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

Left case:

$$\{P * \lceil \forall x.\ Q\ x \rceil\}$$
$$e$$
$$\{\lceil \forall x.\ Q\ x \rceil\}$$

Right case:

$$\{P * \lceil \exists x.\ \neg\ Q\ x \rceil\}$$
$$\{\exists x.\ P * \lceil \neg\ Q\ x \rceil\}$$
$$e$$
$$\{\exists x.\ \lceil Q\ x \rceil * \lceil \neg\ Q\ x \rceil\}$$

Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\,x \rceil\}$
Goal: $\{P\}\ e\ \{\_.\ \lceil \forall x.\ Q\,x \rceil\}$

$$\{P\}$$

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

Left case:

$$\{P * \lceil \forall x.\ Q\ x \rceil\}$$
$$e$$
$$\{\lceil \forall x.\ Q\ x \rceil\}$$

Right case:

$$\{P * \lceil \exists x.\ \neg\ Q\ x \rceil\}$$
$$\{\exists x.\ P * \lceil \neg\ Q\ x \rceil\}$$
$$e$$
$$\{\exists x.\ \lceil Q\ x \rceil * \lceil \neg\ Q\ x \rceil\}$$
$$\{\ \mathit{false}\ \}$$

Proof outline — infinitary, non-dependent case

Hypothesis: $\forall x.\ \{P\}\ e\ \{\_.\ \lceil Q\,x \rceil\}$
Goal: $\{P\}\ e\ \{\_.\ \lceil \forall x.\ Q\,x \rceil\}$

$$\{P\}$$

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

Case $(\forall x.\ Q\ x)$:

$$\{P * \lceil \forall x.\ Q\ x \rceil\}$$
$$e$$
$$\{\lceil \forall x.\ Q\ x \rceil\}$$

Case $(\exists x.\ \neg\ Q\ x)$:

$$\{P * \lceil \exists x.\ \neg\ Q\ x \rceil\}$$
$$\{\exists x.\ P * \lceil \neg\ Q\ x \rceil\}$$
$$e$$
$$\{\exists x.\ \lceil Q\ x \rceil * \lceil \neg\ Q\ x \rceil\}$$
$$\{\ \mathit{false}\ \}$$
$$\{\ \lceil \forall x.\ Q\ x \rceil\ \}$$

The infinitary, dependent case

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

$$\frac{\forall x.\ \{P\}\ e\ \{y.\ \lceil Q\, x\, y \rceil\}}{\{P\}\ e'\ \{y.\ \lceil \forall x. Q\, x\, y \rceil\}}\ \textsc{Infinitary, Dependent}$$

Because of this, e' in the conclusion is a copy of e instrumented with *newProph* and *resolveProph* instructions. (Ouch.)

Contents

- 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:

$$\mathcal{E} \text{ is monotone } \Rightarrow$$
$$\{\textit{eqs implements flip}\,\mathcal{E}\}$$
$$\textit{lfp eqs}$$
$$\{\textit{get}.\ \textit{get implements } \bar{\mu}\mathcal{E}\}$$

where $\bar{\mu}\mathcal{E}$ is *the optimal least fixed point of* $\mathcal{E}$.

Limitations

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*.

Contents

Le Charlier, Baudouin and Pascal Van Hentenryck (1992). *A Universal Top-Down Fixpoint Algorithm*. Technical Report CS-92-25. Brown University.

Vergauwen, Bart and Johan Lewi (1994). *Efficient Local Correctness Checking for Single and Alternating Boolean Equation Systems*. In: *International Colloquium on Automata, Languages and Programming*. Vol. 820. Lecture Notes in Computer Science. Springer, pp. 304–315.

Fecht, Christian and Helmut Seidl (1999). *A Faster Solver for General Systems of Equations*. In: *Science of Computer Programming* 35.2–3, pp. 137–162.

Longley, John (1999). *When is a Functional Program Not a Functional Program?* In: *International Conference on Functional Programming (ICFP)*, pp. 1–7.

References II

Pottier, Francois (2009). *Lazy Least Fixed Points in ML*. Unpublished.

Hofmann, Martin et al. (2010a). *Verifying a Local Generic Solver in Coq*. In: *Static Analysis Symposium (SAS)*. Vol. 6337. Lecture Notes in Computer Science. Springer, pp. 340–355.

Hofmann, Martin et al. (2010b). *What Is a Pure Functional?* In: *International Colloquium on Automata, Languages and Programming*. Vol. 6199. Lecture Notes in Computer Science. Springer, pp. 199–210.

Bauer, Andrej et al. (2013). *On Monadic Parametricity of Second-Order Functionals*. In: *Foundations of Software Science and Computation Structures (FOSSACS)*. Vol. 7794. Lecture Notes in Computer Science. Springer, pp. 225–240.