Staged Specification Logic for Verifying Higher-Order Imperative Programs

Darius Foo, Yahui Song, Wei-Ngan Chin

FM 2024, Milan, Italy

13 September

Challenges: Effectful Higher-order Functions

- Programs today are rife with effectful higher-order functions, but (automated) verifier support for them varies greatly
  - Pure only, e.g. Dafny, Why3, Cameleer
  - Type system guarantees, e.g. Creusot, Prusti
  - Interactive, e.g. Iris, CFML, Pulse/Steel (F*)

- Even when they are supported, specifications are often *imprecise*

- Is there a *precise* and *general* way to support effectful higher-order functions in *automated* program verifiers?

Motivating Example

```
let rec foldr f a l =
  match l with
  | [] => a
  | h :: t =>
    f h (foldr f a t)
```

- *f* is *effectful*: it may have state, exceptions, algebraic effects...
- How do we specify *foldr* in a way that allows the following client to be verified?

```
let count = ref 0 in
foldr (fun c t -> incr count; c + t) 0 xs
```

Specification in Iris

Some clients may want to operate only on certain kinds of lists

f must preserve the invariant

$$\forall P, Inv, f, xs, l. \left\{ \begin{array}{l} (\forall x, a', ys. \{P\ x * Inv\ ys\ a'\}\ f(x, a')\ \{r.\ Inv\ (x{::}ys)\ r\}) \\ * isList\ l\ xs * all\ P\ xs * Inv\ []\ a \end{array} \right\}$$

foldr should not change the list

$$foldr\ f\ a\ l$$

$$\{r.\ isList\ l\ xs * Inv\ xs\ r\}$$

(Separation logic) property relating suffix of input list traversed to result

```
let rec foldr f a l =
  match l with
  | [] => a
  | h :: t =>
    f h (foldr f a t)
```

2

The use of abstract properties

$$\forall P, Inv, f, xs, l. \left\{ \begin{array}{l} (\forall x, a', ys.\, \{P\ x * Inv\ ys\ a'\}\ f(x, a')\ \{r.\ Inv\ (x{::}ys)\ r\}) \\ * \, isList\ l\ xs * all\ P\ xs * Inv\ []\ a \end{array} \right\}$$
$$foldr\ f\ a\ l$$
$$\{r.\ isList\ l\ xs * Inv\ xs\ r\}$$

- The specification commits to an *abstraction* of *f*'s behavior
- This abstraction may not be precise enough for a given client

The specification of foldr is higher-order in the sense that it involves nested Hoare triples (here in the precondition). The reason being that foldr takes a function f as argument, hence we can't specify foldr without having some knowledge or specification for the function f. Different clients may instantiate foldr with some very different functions, hence it can be hard to give a specification for f that is reasonable and general enough to support all these choices. In particular knowing when one has found a good and provable specification can be difficult in itself.

https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf (pg 32)

Problem 1: mutating the list

$$\forall P, Inv, f, xs, l. \left\{ \begin{array}{l} (\forall x, a', ys. \{P\ x * Inv\ \ ys\ \ a'\}\ f(x, a')\ \{r.\ Inv\ (x{::}ys)\ r\}) \\ * \, isList\ l\ \ xs * all\ \ P\ \ xs * Inv\ [] \ a \end{array} \right\}$$

$$foldr\ \ f\ \ a\ \ l$$

$$\{r.\ isList\ l\ \ xs * Inv\ \ xs\ \ r\}$$

```
let foldr_ex1 l = foldr (fun x r -> let v = !x
                                     in x := v+1; v+r) l 0
```

- To specify the list mutation, we would need to state *isList l xs’*
- *Inv xs r* tells us nothing about *xs’*

4

Problem 2: stronger precondition

$$\forall P, Inv, f, xs, l. \left\{ \begin{array}{l} (\forall x, a', ys. \{P\ x * Inv\ ys\ a'\}\ f(x, a')\ \{r.\ Inv\ (x{::}ys)\ r\}) \\ * \mathit{isList}\ l\ xs * all\ P\ xs * Inv\ []\ a \end{array} \right\}$$

$$\mathit{foldr}\ f\ a\ l$$

$$\{r.\ \mathit{isList}\ l\ xs * Inv\ xs\ r\}$$

```
let foldr_ex2 l = foldr (fun x r -> assert(x+r>=0);x+r) l 0
```

- This function argument relies on a property concerning intermediate results
- *P* constrains individual elements only
- *Inv* tells us about *r*, but not *x*
- It's possible to assume something stronger here (x ≥ 0 ∧ r ≥ 0), but it's awkward in general to decompose the property into two parts

Problem 3: effects outside metalogic

$$\forall P, Inv, f, xs, l. \left\{ \begin{array}{l} (\forall x, a', ys. \{P\ x * Inv\ ys\ a'\}\ f(x, a')\ \{r.\ Inv\ (x{::}ys)\ r\}) \\ * isList\ l\ xs * all\ P\ xs * Inv\ []\ a \end{array} \right\}$$

$$foldr\ f\ a\ l$$

$$\{r.\ isList\ l\ xs * Inv\ xs\ r\}$$

```
let foldr_ex3 l = foldr (fun x r -> if x>=0 then x+r
                                    else raise Exc()) l 0
```

- *f* must *return* a value to preserve the invariant
- Trying to abstract *f*'s behavior into a predicate of the underlying logic limits expressiveness
- A pure logic (e.g. SMT) cannot abstract over mutation
- Separation logic allows mutation, but not exceptions/effects

Why was abstraction needed?

- It is difficult to represent unknown higher-order effectful calls *precisely* in pre/post specifications
- Idea: generalize *Hoare triples* with ingredients needed

assert/exhale Sequencing

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi ; \varphi \mid f(x, r) \mid \exists y \,.\, \varphi \mid \varphi \vee \varphi$$

assume/inhale (Un)interpreted relation

$$D, P, Q ::= \sigma \wedge \pi \qquad \sigma ::= emp \mid x \mapsto y \mid \sigma * \sigma \mid \ldots$$

Intuition (semantics).

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi; \varphi \mid f(x, r) \mid \exists\, y\, .\, \varphi \mid \varphi \vee \varphi$$

$$\{P\}\ e\ \{r.\ Q\} \ \equiv\ \forall s, s'\, .\, \langle s, e\rangle \longrightarrow \langle s', v\rangle \wedge (s \vDash P) \Rightarrow \langle s', v\rangle \vDash Q$$

$$\{P\}\ e\ \{r.\ Q\} \ \equiv\ \{\mathbf{ens}\, \mathit{emp}\}\ e\ \{\mathbf{req}\, P; \mathbf{ens}[r]\, Q\}$$

$$\{\mathbf{ens}\, \mathit{emp}\}\ e\ \{\varphi\} \ \equiv\ \forall s, s'\, .\, \langle s, e\rangle \longrightarrow \langle s', v\rangle \Rightarrow \langle s, s', v\rangle \vDash \varphi$$

Intuition (reasoning)

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi ; \varphi \mid f(x, r) \mid \exists\, y \,.\, \varphi \mid \varphi \vee \varphi$$

$$\frac{}{\{\ x \mapsto y\ \}\ !x\ \{\ r.\ x \mapsto y \wedge r{=}y\ \}}\ \textbf{SLDeref}$$

$$\frac{}{\{\varphi\}\ !x\ \{\varphi\,;\exists\, y, r\,.\,\mathbf{req}\, x \mapsto y;\ \mathbf{ens}[r]\ x \mapsto y{\wedge}r{=}y\}}\ \textbf{StDeref}$$

Intuition (reasoning)

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi;\varphi \mid f(x,r) \mid \exists\, y\,.\,\varphi \mid \varphi \vee \varphi$$

$$\frac{(\forall y.\ \{\ P_f\ \}\ f(y)\ \{\ r.\ Q_f\ \}) \qquad P \vdash P_f[x/y] * F}{\{\ P\ \}\ f(x)\ \{\ r.\ Q_f[x/y] * F\ \}}\ \textbf{SLApp}$$

$$\frac{}{\{\varphi\}\ f(x)\ \{\varphi\,;\exists\, r\,.\,f(x,r)\}}\ \textbf{StApp}$$

Our solution: *staged logic*

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi; \varphi \mid f(x, r) \mid \exists\, y \,.\, \varphi \mid \varphi \vee \varphi$$

1. *Sequencing* and *uninterpreted relations*
2. *Recursive* formulae
3. *Re-summarization* of recursion/lemmas
4. *Compaction* via biabduction

$\Longrightarrow$ *Defer abstraction* until appropriate

1. Sequencing and uninterpreted relations

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi;\varphi \mid f(x,r) \mid \exists y\,.\,\varphi \mid \varphi \vee \varphi$$

```
let hello f x y =
  x := !x + 1;
  let r = f y in
  let r2 = !x + r in
  y := r2;
  r2
```

$$\begin{aligned}
&\mathit{hello}(f,x,y,\mathit{res}) = \\
&\quad \exists a\,.\,\mathbf{req}\, x \mapsto a;\, \mathbf{ens}\, x \mapsto a{+}1; \\
&\quad \exists r\,.\,\boxed{f(y,r);} \\
&\quad \exists b\,.\ \mathbf{req}\, x \mapsto b * y \mapsto _; \\
&\qquad\quad \mathbf{ens}\, x \mapsto b * y \mapsto \mathit{res} \wedge \mathit{res}{=}b{+}r
\end{aligned}$$

- Uninterpreted relations represent unknown function parameters
- Sequencing allows them to serve as *placeholders* for effects

2. Recursive formulae

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi;\varphi \mid f(x,r) \mid \exists\, y\,.\,\varphi \mid \varphi \vee \varphi$$

```
let rec foldr f a l =
  match l with
  | [] => a
  | h :: t =>
    f h (foldr f a t)
```

$$\begin{aligned} &\mathit{foldr}(f,a,l,\mathit{res}) = \\ &\qquad \mathbf{ens}\ l{=}[]\wedge \mathit{res}{=}a \\ &\quad \vee\ \exists\, r, l_1\,;\ \mathbf{ens}\ l{=}x{::}l_1; \\ &\qquad\qquad \mathit{foldr}(f,a,l_1,r); f(x,r,\mathit{res}) \end{aligned}$$

$$\begin{aligned} &\mathit{foldr}(f,a,l,\mathit{res}) = \\ &\quad \exists\, P, \mathit{Inv}, xs\,.\,\mathbf{req}\ \mathit{List}(l,xs) * \mathit{Inv}([],a) \wedge \mathit{all}(P,xs) \\ &\qquad \wedge f(x,a',r) \sqsubseteq (\exists\, ys\,.\,\mathbf{req}\ \mathit{Inv}(ys,a') \wedge P(x); \mathbf{ens}[r]\ \mathit{Inv}(x{::}ys,r))\,; \\ &\quad \mathbf{ens}[\mathit{res}]\ \mathit{List}(l,xs) * \mathit{Inv}(xs,\mathit{res}) \end{aligned}$$

- The call to *f* can be represented directly, without requiring abstraction
- Recursion is used where needed

3. Re-summarization of recursion/lemmas

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi ; \varphi \mid f(x, r) \mid \exists\, y \,.\, \varphi \mid \varphi \vee \varphi$$

- Recovering abstraction: re-summarization

```
let foldr_sum_state x xs init
```

$$\mathit{foldr\_sum\_state}(x, \mathit{xs}, \mathit{init}, \mathit{res}) = \\ \quad \exists\, i, r \,.\, \mathbf{req}\, x \mapsto i ; \mathbf{ens}\, x \mapsto i{+}r \wedge \mathit{res}{=}r{+}\mathit{init} \wedge \mathit{sum}(\mathit{xs}, r)$$

```
= let g c t = x := !x + c; c + t in foldr g xs init
```

$$\forall x, \mathit{xs}, \mathit{init}, \mathit{res} \,.\, \begin{array}{l} \mathit{foldr}(g, \mathit{xs}, \mathit{init}, \mathit{res}) \\ \sqsubseteq \exists\, i, r \,.\, \mathbf{req}\, x \mapsto i ; \mathbf{ens}\, x \mapsto i{+}r \wedge \mathit{res}{=}r{+}\mathit{init} \wedge r{=}\mathit{sum}(\mathit{xs}) \end{array}$$

3. Re-summarization of recursion/lemmas

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi ; \varphi \mid f(x, r) \mid \exists\, y \,.\, \varphi \mid \varphi \vee \varphi$$

- Recovering abstraction: proving entailments

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{
    \dfrac{
      \dfrac{}{x \mapsto i \vdash \exists\, i \,.\, x \mapsto i * \mathit{emp}}\ \text{SL}
      \qquad
      \dfrac{}{\begin{array}{c} x \mapsto i{+}r{+}h \wedge \mathit{res}{=}h{+}r{+}\mathit{init} \wedge r{=}\mathit{sum}(t) \\ \vdash \exists\, r \,.\, x \mapsto i{+}r \wedge \mathit{res}{=}r{+}\mathit{init} \wedge r{=}\mathit{sum}(h{::}t) \end{array}}\ \text{SL}
    }{\begin{array}{c} \exists\, i \,.\, \mathbf{req}\, x \mapsto i ; \exists\, r, h, t \,.\, \mathbf{ens}\, x \mapsto i{+}r{+}h \wedge \mathit{res}{=}h{+}r{+}\mathit{init} \wedge r{=}\mathit{sum}(t) \wedge \mathit{xs}{=}h{::}t \\ \sqsubseteq \exists\, i \,.\, \mathbf{req}\, x \mapsto i ; \exists\, r \,.\, \mathbf{ens}\, x \mapsto i{+}r \wedge \mathit{res}{=}r{+}\mathit{init} \wedge r{=}\mathit{sum}(\mathit{xs}) \end{array}}\ \textsc{Entail}
   }{\begin{array}{c} \exists\, r_1, h, t \,.\, \mathbf{ens}\, \mathit{xs}{=}h{::}t ; \exists\, i, r \,.\, \mathbf{req}\, x \mapsto i ; \mathbf{ens}\, x \mapsto i{+}r \wedge r_1{=}r{+}\mathit{init} \wedge r{=}\mathit{sum}(t) ; \\ \exists\, b \,.\, \mathbf{req}\, x \mapsto b ; \mathbf{ens}\, x \mapsto b{+}h \wedge \mathit{res}{=}h{+}r_1 \sqsubseteq \ldots \end{array}}\ \textsc{Normalize}
  }{\exists\, r_1, h, t \,.\, \mathbf{ens}\, \mathit{xs}{=}h{::}t ; \exists\, i, r \,.\, \mathbf{req}\, x \mapsto i ; \mathbf{ens}\, x \mapsto i{+}r \wedge r_1{=}r{+}\mathit{init} \wedge r{=}\mathit{sum}(t) ; g(h, r_1, \mathit{res}) \sqsubseteq \ldots}\ \textsc{Unfold}
 }{(\ldots \vee \exists\, r_1, h, t \,.\, \mathbf{ens}\, \mathit{xs}{=}h{::}t ; \mathit{foldr}(g, t, \mathit{init}, r_1) ; g(h, r_1, \mathit{res})) \sqsubseteq \ldots}\ \textsc{Induction}
}{\forall x, \mathit{xs}, \mathit{init}, \mathit{res} \,.\ \begin{array}{l} \mathit{foldr}(g, \mathit{xs}, \mathit{init}, \mathit{res}) \\ \sqsubseteq \exists\, i, r \,.\, \mathbf{req}\, x \mapsto i ; \mathbf{ens}\, x \mapsto i{+}r \wedge \mathit{res}{=}r{+}\mathit{init} \wedge r{=}\mathit{sum}(\mathit{xs}) \end{array}}\ \textsc{Unfold}
$$

4. Compaction via biabduction

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi;\varphi \mid f(x,r) \mid \exists\, y\,.\,\varphi \mid \varphi \vee \varphi$$

$$\frac{\begin{array}{c}\exists\, i\,.\,\mathbf{req}\, x \mapsto i; \exists\, r,h,t\,.\,\mathbf{ens}\, x \mapsto i{+}r{+}h \wedge res{=}h{+}r{+}init \wedge r{=}sum(t) \wedge xs{=}h{::}t \\ \sqsubseteq \exists\, i\,.\,\mathbf{req}\, x \mapsto i; \exists\, r\,.\,\mathbf{ens}\, x \mapsto i{+}r \wedge res{=}r{+}init \wedge r{=}sum(xs)\end{array}}{\begin{array}{c}\exists\, r_1,h,t\,.\,\mathbf{ens}\, xs{=}h{::}t; \exists\, i,r\,.\,\mathbf{req}\, x \mapsto i; \mathbf{ens}\, x \mapsto i{+}r \wedge r_1{=}r{+}init \wedge r{=}sum(t); \\ \exists\, b\,.\,\mathbf{req}\, x \mapsto b; \mathbf{ens}\, x \mapsto b{+}h \wedge res{=}h{+}r_1 \sqsubseteq \ldots\end{array}} \ \textsc{Normalize}$$

$$\frac{D_a * D_1 \vdash D_2 * D_f}{\mathbf{ens}\, D_1; \mathbf{req}\, D_2 \Longrightarrow \mathbf{req}\, D_a; \mathbf{ens}\, D_f}$$

4. Compaction via biabduction

$$\varphi \;::=\; \mathbf{req}\,P \mid \mathbf{ens}[r]\,Q \mid \varphi;\varphi \mid f(x,r) \mid \exists\, y\,.\,\varphi \mid \varphi \vee \varphi$$

$$\dfrac{\begin{array}{c}\exists\, i\,.\,\mathbf{req}\,x \mapsto i; \exists\, r,h,t\,.\,\mathbf{ens}\,x \mapsto i{+}r{+}h \wedge res{=}h{+}r{+}init \wedge r{=}sum(t) \wedge xs{=}h{::}t \\ \sqsubseteq \exists\, i\,.\,\mathbf{req}\,x \mapsto i; \exists\, r\,.\,\mathbf{ens}\,x \mapsto i{+}r \wedge res{=}r{+}init \wedge r{=}sum(xs)\end{array}}{\begin{array}{c}\exists\, r_1,h,t\,.\,\mathbf{ens}\,xs{=}h{::}t; \exists\, i,r\,.\,\mathbf{req}\,x \mapsto i; \mathbf{ens}\,x \mapsto i{+}r \wedge r_1{=}r{+}init \wedge r{=}sum(t); \\ \exists\, b\,.\,\mathbf{req}\,x \mapsto b; \mathbf{ens}\,x \mapsto b{+}h \wedge res{=}h{+}r_1 \sqsubseteq \ldots\end{array}}\;\textsc{Normalize}$$

$$\dfrac{(b{=}i{+}r) * x \mapsto i{+}r \wedge D \vdash x \mapsto b * D}{\mathbf{ens}\,x \mapsto i{+}r \wedge D; \mathbf{req}\,x \mapsto b \implies \mathbf{req}\,b{=}i{+}r; \mathbf{ens}\,D}$$

Problem 1: mutating the list

```
let foldr_ex1 l = foldr (fun x r -> let v = !x
                                     in x := v+1; v+r) l 0
```

$$
\begin{aligned}
\mathit{foldr\_ex1}\,(l, \mathit{res}) \sqsubseteq\; & \exists\, xs\,.\,\mathbf{req}\ \mathit{List}(l, xs); \\
& \exists\, ys\,.\,\mathbf{ens}\ \mathit{List}(l, ys) \wedge \mathit{mapinc}(xs){=}ys \wedge \mathit{sum}(xs){=}\mathit{res}
\end{aligned}
$$

- An invariant is not needed to specify the function argument
- We can directly use a shape predicate, with value described by a pure function

Problem 2: stronger precondition

```
let foldr_ex2 l = foldr (fun x r -> assert(x+r>=0);x+r) l 0
```

$$foldr\_ex2(l, res) \sqsubseteq \mathbf{req}\ allSPos(l);\ \mathbf{ens}\ sum(l){=}res$$

- We can directly use a predicate on l to require that all suffix-sums are positive

Problem 3: effects outside metalogic

```
let foldr_ex3 l = foldr (fun x r -> if x>=0 then x+r
                                     else raise Exc()) l 0
```

$$foldr\_ex3(l, res) \sqsubseteq \mathbf{ens}\ allPos(l) \wedge sum(l) = res \ \vee\ (\mathbf{ens}[\_]\ \neg allPos(l); Exc())$$

- An exception can be modelled as an *interpreted* relation (more in ICFP 2024)
- We do not delegate effects to the underlying separation logic

Implementation & Evaluation

Benchmark	Heifer				Cameleer [21]			Prusti [27]		
	LoC	LoS	T	T_P	LoC	LoS	T	LoC	LoS	T
map	13	11	0.66	0.58	10	45	1.25		-	
map_closure	18	7	1.06	0.77		✗			-	
fold	23	12	1.06	0.87	21	48	8.08		-	
fold_closure	23	12	1.25	0.89		✗			-	
iter	11	4	0.40	0.32		✗			-	
compose	3	1	0.11	0.09	2	6	0.05		-	
compose_closure	23	4	0.44	0.32		✗			✗	
closure [24]	27	5	0.37	0.27		✗		13	11	6.75
closure_list	7	1	0.15	0.09		✗			-	
applyN	6	1	0.19	0.17	12	13	0.37		-	
blameassgn [11]	14	6	0.31	0.28		✗		13	9	6.24
counter [16]	16	4	0.24	0.18		✗		11	7	6.37
lambda	13	5	0.25	0.22		✗			-	
	197	73			45	112		37	27	

Table 1: A Comparison with Cameleer and Prusti. (Programs that are natively inexpressible are marked with "✗". Programs that cannot be reproduced from Prusti's artifact [1] are marked with "-". We use T to denote the total verification time (in seconds) and T_P to record the time spent on the external provers.)

- 5K LoC, OCaml 5
- Small but representative examples
- Reasonably low verification time
- 0.37 spec : code
- Feasibility & increased expressiveness over existing systems

Conclusion

$$\varphi \ ::= \ \mathbf{req}\, P \mid \mathbf{ens}[r]\, Q \mid \varphi ; \varphi \mid f(x, r) \mid \exists y \,.\, \varphi \mid \varphi \vee \varphi$$

- Staged logic for effectful higher-order programs
  1. *Sequencing* and *uninterpreted relations*
  2. *Recursive* formulae
  3. *Re-summarization* of recursion/lemmas
  4. *Compaction* via biabduction

  $\Longrightarrow$ *Defer abstraction* until appropriate
- Heifer – a new automated verifier
  - https://github.com/hipsleek/heifer

Thanks for listening!

*Higher-order Effectful Imperative Function Entailments and Reasoning