Mechanising Staged Logic

Darius Foo, Yahui Song, Wei-Ngan Chin

SG PL Summit

4 December 2024

Staged logic

- A new, alternative program logic formulation
- Automated (SMT-based) verification
- Effectful higher-order programs

Staged Specification Logic for Verifying Higher-Order Imperative Programs

Darius Foo(✉)[0000-0002-3279-5827], Yahui Song[0000-0002-9760-5895], and Wei-Ngan Chin[0000-0002-9660-5682]

School of Computing, National University of Singapore, Singapore
{dariusf,yahuis,chinwn}@comp.nus.edu.sg

An effectful higher-order program

```
let x = ref [] in
foldr (fun c t -> x := c :: !x; c + t) xs 0
```

A specification we would like to give it

$$\forall x\ a,\ \{x \mapsto a\}$$

```
foldr (fun c t -> x := c :: !x; c + t) xs 0
```

$$\{res.\ x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]\}$$

How do we prove it automatically?

The traditional approach

- Parameterise specification of foldr over invariants/properties

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

f must preserve the invariant

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

$$foldr\ f\ a\ l$$

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

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

The traditional approach

- Parameterise specification of foldr over invariants/properties
- Automation is difficult
  - How to infer invariants/properties to be supplied at call sites?
  - How to infer specification? Clients require different parameterisations

Staged logic

- Natively represent effectful behavior in the logic
- The proof can then be done directly by induction
  - Enabling existing techniques for automated inductive proof [Sun 24]

[Sun 24] “Proving Functional Program Equivalence via Directed Lemma Synthesis” Sun et al. FM 2024

The rest of this talk

- What are the primitives we need? How do proofs work? (Part I)
- How do we mechanise the proof steps in Coq? (Part II)

Syntax of staged logic

Retain ordering of
unknown calls/effects

Program states

$$\varphi ::= \mathbf{req}\,\sigma \mid \mathbf{ens}[r]\,\sigma \mid f(v, r) \mid \varphi; \varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \forall x.\varphi$$

Obligation*

Assumption*

*Flips depending on which side of a call/entailment we're on

Syntax of staged logic

$$\varphi ::= \mathbf{req}\,\sigma \mid \mathbf{ens}[r]\,\sigma \mid f(v, r) \mid \varphi; \varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \forall x.\varphi$$

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

$$e; f(a, r) ::: \mathbf{req}\,P; \mathbf{ens}[r]\,Q; f(a, r)$$

foldr

```
let foldr f init xs =
  match xs with
  | [] => init
  | h :: t =>
    f h (foldr f init t)
```

$$
\begin{aligned}
&\mathit{foldr}(f, \mathit{init}, \mathit{xs}, \mathit{res}) = \\
&\qquad \mathbf{ens}[\mathit{res}]\ \mathit{xs}{=}[\,] \wedge \mathit{res}{=}\mathit{init} \\
&\quad \vee\ \exists\, h, t\,.\ \mathbf{ens}\ \mathit{xs}{=}h{::}t; \\
&\qquad\qquad \exists\, r\,.\ \mathit{foldr}(f, \mathit{init}, t, r); f\,(h, r, \mathit{res})
\end{aligned}
$$

Giving a specification: entailment

$$foldr(g, init, xs, res)$$
$$\sqsubseteq \textbf{req}\, x \mapsto a; \textbf{ens}[res]\; x \mapsto (xs +\!+\; a) * [res = sum\ xs]$$

where `(fun c t -> x := c :: !x; c + t)` $:: g$

A proof strategy

1. Choose argument to perform induction on
2. Unfold non-recursive predicates
3. Rewrite using lemmas/induction hypothesis
4. Normalize, reaching the form

$$\left(\mathbf{req}\,\sigma; \mathbf{ens}\,\sigma; f\,(a, r);\right)^{*} \mathbf{req}\,\sigma; \mathbf{ens}[r]\;\sigma$$

5. Dispatch proof obligations using entailment rules

The proof, very briefly

By well-founded induction on xs

$$
\begin{aligned}
& foldr(g, init, xs, res) \\
& \sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\, x \mapsto (xs +\!+\, a) * [res = sum\ xs]
\end{aligned}
$$

The proof, very briefly

Unfold foldr (and focus on the recursive case)

$$
\begin{aligned}
& foldr(g, init, xs, res) \\
& \sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\, x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]
\end{aligned}
$$

The proof, very briefly

Unfold foldr (and focus on the recursive case)

$$\mathbf{ens}\,(xs = h :: t); foldr(f, init, t, r); f\,(h, r, res)$$
$$\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\, x \mapsto (xs \mathbin{+\!\!+} a) * [res = sum\ xs]$$

The proof, very briefly

Unfold foldr (and focus on the recursive case)

$$\begin{aligned}
&\mathbf{ens}\,(xs = h :: t);\\
&foldr(f, init, t, r);\\
&f\,(h, r, res)\\
&\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\, x \mapsto (xs +\!\!+\, a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

Rewrite using the IH

$$\boxed{t} \leq \boxed{xs}$$

$$\begin{aligned}
&\mathbf{ens}\ (\boxed{xs} = h :: \boxed{t});\\
&foldr(f, init, \boxed{t}, r);\\
&f(h, r, res)\\
&\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\ x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

Rewrite using the IH

$$
\begin{aligned}
&\mathbf{ens}\,(xs = h :: t);\\
&\forall a_1, \mathbf{req}\, x \mapsto a_1; \mathbf{ens}\,(x \mapsto (t \mathbin{+\!\!+} a_1) * [r = sum\ t]);\\
&f\,(h, r, res)\\
&\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\, x \mapsto (xs \mathbin{+\!\!+} a) * [res = sum\ xs]
\end{aligned}
$$

The proof, very briefly

Unfold f

$$
\begin{aligned}
&\mathbf{ens}\,(xs = h :: t);\\
&\forall a_1, \mathbf{req}\, x \mapsto a_1; \mathbf{ens}\,(x \mapsto (t \mathbin{++} a_1) * [r = sum\ t]);\\
&f\,(h, r, res)\\
&\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\, x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]
\end{aligned}
$$

The proof, very briefly

Unfold f

$$\begin{aligned}
&\mathbf{ens}\,(xs = h :: t);\\
&\forall a_1, \mathbf{req}\, x \mapsto a_1; \mathbf{ens}\,(x \mapsto (t \mathbin{++} a_1) * [r = sum\ t]);\\
&\forall z, \mathbf{req}\, x \mapsto z; \mathbf{ens}[res]\, x \mapsto (h :: z) * [res = h + r]\\
&\sqsubseteq\ \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\, x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

Normalise

$$\frac{H_A * H_1 \vdash H_2 * H_F}{\textbf{ens}\, H_1; \textbf{req}\, H_2 \sqsubseteq \textbf{req}\, H_A; \textbf{ens}\, H_F}\ \textbf{NormEnsReq}$$

$$\begin{aligned}
&\textbf{ens}\,(xs = h :: t);\\
&\forall a_1, \textbf{req}\, x \mapsto a_1; \textbf{ens}\,(x \mapsto (t \mathbin{++} a_1) * [r = sum\ t]);\\
&\forall z, \textbf{req}\, x \mapsto z; \textbf{ens}[res]\ x \mapsto (h :: z) * [res = h + r]\\
&\sqsubseteq \textbf{req}\, x \mapsto a; \textbf{ens}[res]\ x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

Normalise

$$\big(z = (t ++ a_1)\big) * \big(x \mapsto (t ++ a_1) * [r = sum\ t]\big) \vdash \big(x \mapsto z\big) * \big([r = sum\ t]\big)$$

$$\begin{aligned}
&\mathbf{ens}\ (xs = h :: t);\\
&\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ (x \mapsto (t ++ a_1) * [r = sum\ t]);\\
&\forall z, \mathbf{req}\ x \mapsto z; \mathbf{ens}[res]\ x \mapsto (h :: z) * [res = h + r]\\
&\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}[res]\ x \mapsto (xs ++ a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

Normalise

$$\big(z = (t \mathbin{++} a_1)\big) * \big(x \mapsto (t \mathbin{++} a_1) * [r = sum\ t]\big) \vdash \big(x \mapsto z\big) * \big([r = sum\ t]\big)$$

$$\begin{aligned}
&\mathbf{ens}\ (xs = h :: t);\\
&\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ (x \mapsto (t \mathbin{++} a_1) * [r = sum\ t]);\\
&\forall z, \mathbf{req}\ x \mapsto z; \mathbf{ens}[res]\ x \mapsto (h :: z) * [res = h + r]\\
&\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}[res]\ x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

Normalise

$$(z = (t ++ a_1)) * (x \mapsto (t ++ a_1) * [r = sum\ t]) \vdash (x \mapsto z) * ([r = sum\ t])$$

$$\begin{aligned}
&\forall a_1, \textbf{req}\, x \mapsto a_1; \\
&\textbf{ens}[res]\, x \mapsto (h :: (t ++ a_1)) * \\
&\qquad [res = h + r \wedge r = sum\ t \wedge xs = h :: t] \\
&\sqsubseteq \textbf{req}\, x \mapsto a; \textbf{ens}[res]\, x \mapsto (xs ++ a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

We have reached normal form

$$
\begin{aligned}
&\forall a_1, \mathbf{req}\, x \mapsto a_1;\\
&\mathbf{ens}[res]\; x \mapsto (h :: (t \mathbin{++} a_1)) *\\
&\qquad [res = h + r \wedge r = sum\ t \wedge xs = h :: t]\\
&\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\; x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]
\end{aligned}
$$

The proof, very briefly

Contravariance of req

$$\frac{H_2 \vdash H_1}{\mathbf{req}\, H_1 \sqsubseteq \mathbf{req}\, H_2}\ \textbf{EntailsReq}$$

$$\forall a_1, \mathbf{req}\, x \mapsto a_1;$$
$$\mathbf{ens}[res]\ x \mapsto (h :: (t \mathbin{++} a_1)) *$$
$$[res = h + r \land r = sum\ t \land xs = h :: t]$$
$$\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\ x \mapsto (xs \mathbin{++} a) * [res = sum\ xs]$$

The proof, very briefly

Contravariance of req

$$x \mapsto a \vdash x \mapsto a$$

$$\begin{aligned}
&\forall a_1, \mathbf{req}\, x \mapsto a_1; \\
&\mathbf{ens}[res]\; x \mapsto (h :: (t \mathbin{+\!\!+} a_1)) * \\
&\qquad [res = h + r \land r = sum\ t \land xs = h :: t] \\
&\sqsubseteq \mathbf{req}\, x \mapsto a; \mathbf{ens}[res]\; x \mapsto (xs \mathbin{+\!\!+} a) * [res = sum\ xs]
\end{aligned}$$

The proof, very briefly

Contravariance of req

$$
\begin{aligned}
&\mathbf{ens}[res]\; x \mapsto (h :: (t +\!+\; a)) * \\
&\qquad [res = h + r \wedge r = sum\; t \wedge xs = h :: t] \\
&\sqsubseteq \mathbf{ens}[res]\; x \mapsto (xs +\!+\; a) * [res = sum\; xs]
\end{aligned}
$$

The proof, very briefly

Covariance of ens

$$\frac{Q_1 \vdash Q_2}{\mathbf{ens}\, Q_1 \sqsubseteq \mathbf{ens}\, Q_2}\ \textbf{EntailsEns}$$

$$\begin{aligned}
&\mathbf{ens}[\mathit{res}]\; x \mapsto (h :: (t \mathbin{++} a)) * \\
&\qquad [\mathit{res} = h + r \land r = \mathit{sum}\ t \land \mathit{xs} = h :: t] \\
&\sqsubseteq \mathbf{ens}[\mathit{res}]\; x \mapsto (\mathit{xs} \mathbin{++} a) * [\mathit{res} = \mathit{sum}\ \mathit{xs}]
\end{aligned}$$

The proof, very briefly

Separation logic entailment

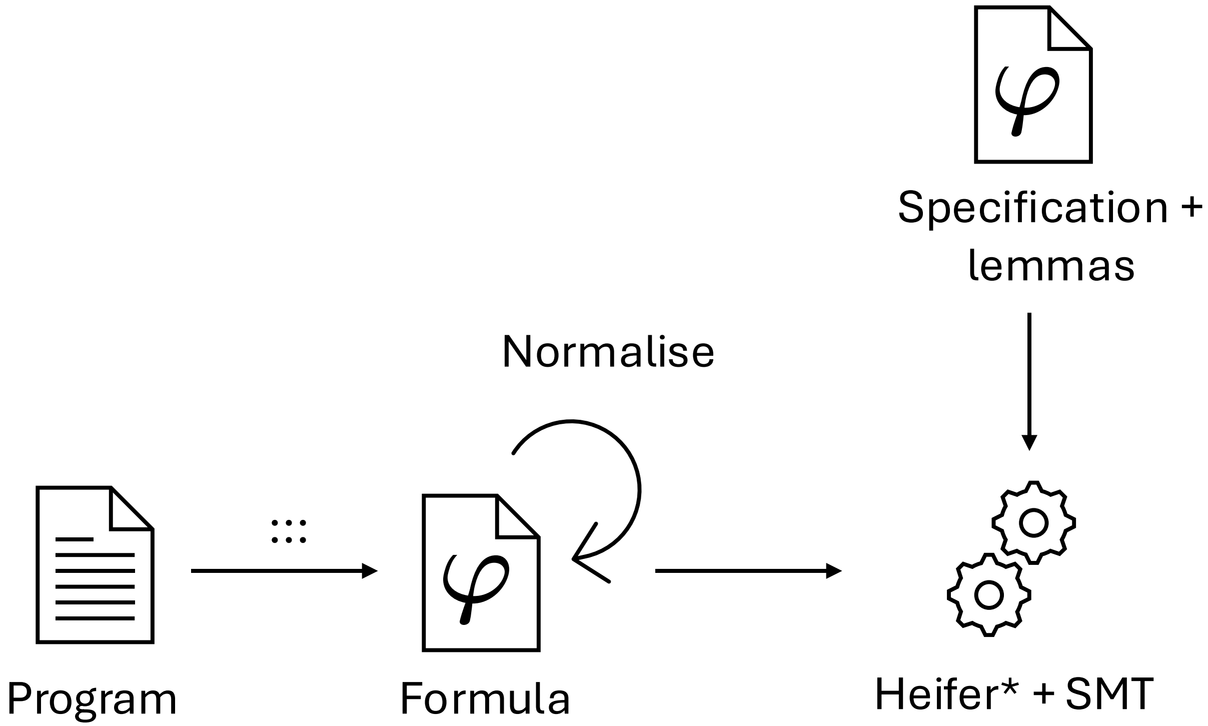
$$
\begin{aligned}
& x \mapsto (h :: (t +\!\!+\, a)) * \\
& \qquad [res = h + r \wedge r = sum\ t \wedge xs = h :: t] \\
& \vdash x \mapsto (xs +\!\!+\, a) * [res = sum\ xs]
\end{aligned}
$$

The proof, very briefly

SMT (and some properties of append and cons)

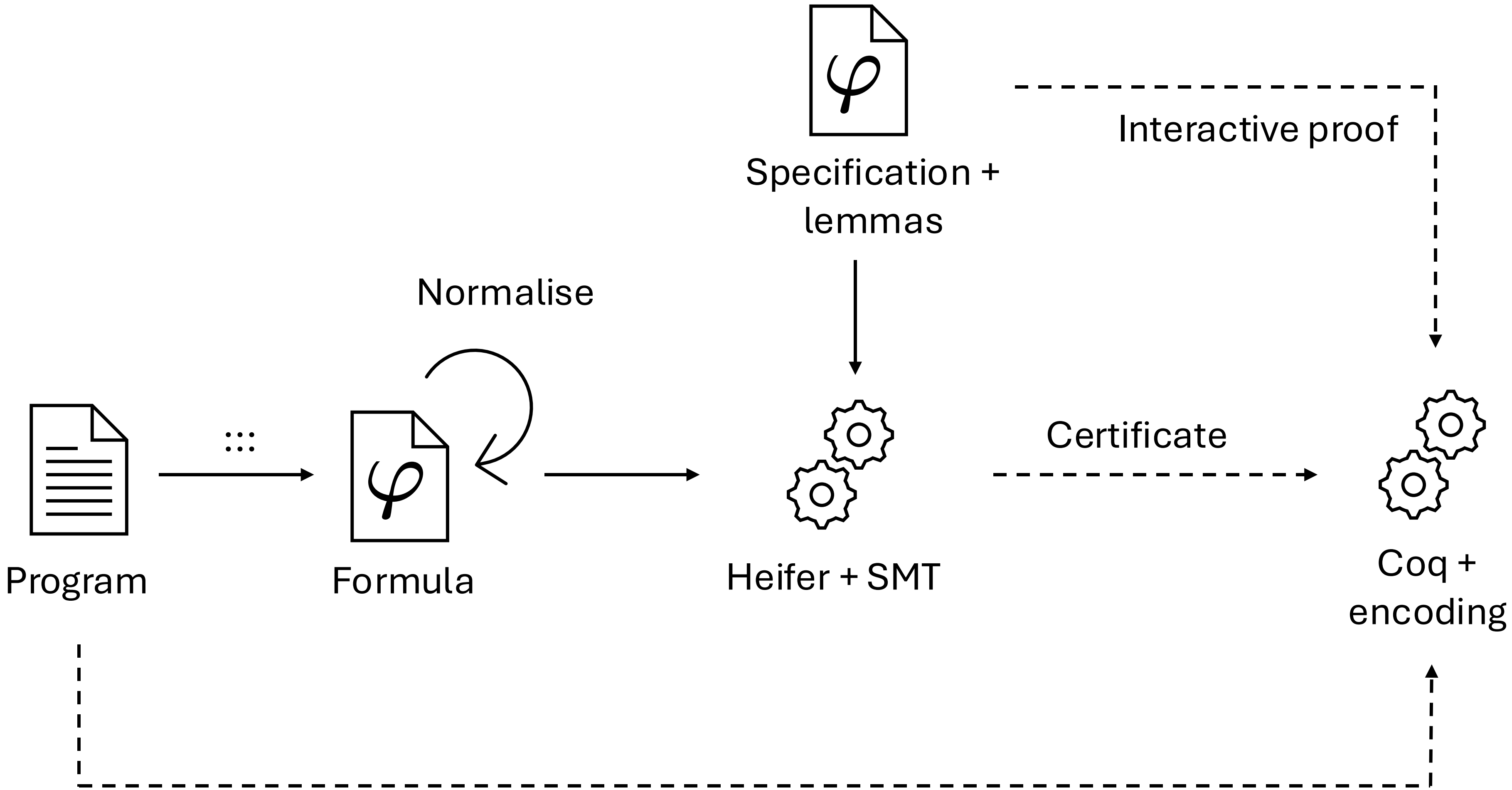
$$
\begin{aligned}
&res = h + (sum\ t) \land xs = h :: t \\
&\Rightarrow h :: (t \mathbin{++} a) = xs \mathbin{++} a \land res = sum\ xs
\end{aligned}
$$

The workflow



*Higher-order Effectful Imperative Function Entailments & Reasoning

The workflow we would like



What we would like

How to encode $\mathcal{E} \vdash \varphi \sqsubseteq \varphi$?

Coq sequent

```
▼Goal (1)
 xs : list val
 IH : forall y : list val,
      list_sub y xs →
      forall res0 : val,
      foldr_env ⊢ "foldr"$(vtup (vstr "f") (vtup (vint 0) (vlist y)), res0)
      ⊑ ∀ (x : loc) (a : list val),
          req (x ⇝ vlist a)
            (ens_ (x ⇝ vlist (y ++ a) \* \[res0 = vint (sum (to_int_list y))]))
 res : val
 ────────────────────────────────────────
  foldr_env
  ⊢ ∃ (x : int) (l1 : list val),
      ens_ \[xs = vint x :: l1];;
      (∃ r : val,
         ("foldr"$(vtup (vstr "f") (vtup (vint 0) (vlist l1)), r));;
         ("f"$(vtup (vint x) r, res)))
  ⊑ ∀ (x : loc) (a : list val),
      req (x ⇝ vlist a)
        (ens_ (x ⇝ vlist (xs ++ a) \* \[res = vint (sum (to_int_list xs))]))
```

Staged logic sequent

Semantics of staged logic

$h \vDash H$

$h, v \vDash Q$

$$\mathcal{E}, h_1, h_2, v \vDash \varphi$$

heaps result

$$\mathcal{E}, h_1, h_2, v \vDash \mathbf{req}\, P\, \varphi \;\; \textit{if}$$
$$\forall h_p\; h_r, (h_p \vDash P \text{ and } h_1 = h_p \circ h_r) \Rightarrow \mathcal{E}, h_r, h_2, v \vDash \varphi$$

$$\mathcal{E}, h_1, h_2, v \vDash \mathbf{ens}\, Q \;\; \textit{if}$$
$$\exists h_3, (h_3, v \vDash Q) \text{ and } h_2 = h_1 \circ h_3$$

Internalization of the operational behaviour of heap entailment

Semantics of staged logic

An environment of unknown functions

$$\mathcal{E}, h_1, h_2, v \vDash \varphi$$

heaps

result

$$\mathcal{E}, h_1, h_2, v \vDash f(a, r) \;\; \mathit{if}$$
$$\qquad \mathcal{E}, h_1, h_2, v \vDash \mathcal{E}[f](a, r)$$

Encoding staged logic

An environment of unknown functions

$$\mathcal{E}, h_1, h_2, v \vDash \varphi$$

heaps result

Separation logic: `heap -> Prop`

Unfortunately, a direct shallow embedding would be impredicative

HOAS encoding, enabling substitution

```
Definition ufun := val -> val -> phi.

Definition phi := map var ufun -> heap -> heap -> val -> Prop.
```

Encoding staged logic

An environment of unknown functions

$$\mathcal{E}, h_1, h_2, v \vDash \varphi$$

heaps result

Separation logic: `heap -> Prop`

Use a deep embedding and interpretation function

```
Inductive phi : Type :=
  | req : hprop -> phi -> phi
  | ens : (val -> hprop) -> phi
  | seq : phi -> phi -> phi
  | unk : var -> val -> val -> phi
  ...
```

```
Definition ufun := val -> val -> phi.

Inductive satisfies :
  map var ufun ->
  heap -> heap -> val -> phi -> Prop := ...
```

Encoding staged logic

Entailment: $\varphi \sqsubseteq \varphi$

We use a semantic definition:

```
Definition entails (f1 f2:phi) : Prop :=
  forall env h1 h2 R,
    satisfies env h1 h2 R f1 ->
      satisfies env h1 h2 R f2.
```

Lemmas about entailment can be stated and proved directly.

$$\frac{H_1 \vdash H_2}{\textbf{ens}\, H_1 \sqsubseteq \textbf{ens}\, H_2}\ \textbf{EntailsEns}$$

```
Lemma entails_ens : forall H1 H2,
  H1 ==> H2 ->
  entails (ens H1) (ens H2).
```

Encoding staged logic

Entailment sequent: $\mathcal{E} \vdash \varphi \sqsubseteq \varphi$

Parameterised over the environment, supporting rules such as:

$$\frac{F = \mathcal{E}(f)}{\mathcal{E} \vdash f(a, r) \sqsubseteq F(a, r)}\ \textbf{EntailsUnfold}$$

What we would like

How to encode $\mathcal{E} \vdash \varphi \sqsubseteq \varphi$? ✅

```
▼Goal (1)
 xs : list val
 IH : forall y : list val,
      list_sub y xs →
      forall res0 : val,
      foldr_env ⊢ "foldr"$(vtup (vstr "f") (vtup (vint 0) (vlist y)), res0)
      ⊑ ∀ (x0 : loc) (a : list val),
          req (x0 ⤳ vlist a)
            (ens_ (x0 ⤳ vlist (y ++ a) \* \[res0 = vint (sum (to_int_list y))]))
 res : val
 x : int
 l1 : list val
 H : xs = vint x :: l1
 r : val
 ________________________________________________________________________
  foldr_env
  ⊢ ("foldr"$(vtup (vstr "f") (vtup (vint 0) (vlist l1)), r));;
    ("f"$(vtup (vint x) r, res))
  ⊑ ∀ (x0 : loc) (a : list val),
      req (x0 ⤳ vlist a)
        (ens_ (x0 ⤳ vlist (xs ++ a) \* \[res = vint (sum (to_int_list xs))]))
```

Rewrite

Unfold

Rewriting

- We use Coq's *setoid rewriting*, with entails as the "equivalence" relation

- entails must be shown to be *proper* in both arguments

$$\frac{\varphi_1 \sqsubseteq \varphi_2 \qquad \varphi_1 \overset{\text{Contravariance}}{\sqsupseteq} \varphi_3 \qquad \varphi_2 \underset{\text{Covariance}}{\sqsubseteq} \varphi_4}{\varphi_3 \sqsubseteq \varphi_4}\ \textbf{ProperEntails}$$

Rewriting

- This can be specified by providing the following typeclass instance

```
#[global]
Instance Proper_entails : Proper
  (flip entails ====> entails ====> impl)
  entails.
```

What we would like

Rewriting ✅

$$\frac{\mathcal{E} \vdash \mathbf{ens}\,\sigma;\varphi_1 \sqsubseteq \varphi_2}{\mathcal{E} \vdash \varphi_1 \sqsubseteq \mathbf{req}\,\sigma;\varphi_2}\ \textbf{EntailsReqR}$$

```
foldr_env
⊢ req (x0 ⤳ vlist a)
    (ens_ (x0 ⤳ vlist (l1 ++ a) \* \[r = vint (sum (to_int_list l1))]);;
     g (vtup (vint x) r) res)
⊑ req (x0 ⤳ vlist a)
    (ens_ (x0 ⤳ vlist (xs ++ a) \* \[res = vint (sum (to_int_list xs))]))
```

Can be introduced into
the "spatial context"

What we would like

The spatial context, or the “current state” in symbolic-execution style verifiers

The precondition of the next “call”, discharged via biabduction

```
foldr_env
⊢ ens_ (x0 ⤳ vlist a);;
  req (x0 ⤳ vlist a)
    (ens_ (x0 ⤳ vlist (l1 ++ a) \* \[r = vint (sum (to_int_list l1))]);;
     g (vtup (vint x) r) res)
⊑ ens_ (x0 ⤳ vlist (xs ++ a) \* \[res = vint (sum (to_int_list xs))])
```

The postcondition of the next “call”

The final proof obligation

What we would like

"Symbolic execution" using biabduction ✅

```
foldr_env
⊢ ens_ (x0 ⤳ vlist (vint x :: l1 ++ a) \* \[res = vint (x + sum (to_int_list l1))])
⊑ ens_ (x0 ⤳ vlist (xs ++ a) \* \[res = vint (sum (to_int_list xs))])
```

What we would like

```
x0 ⇝ vlist (vint x :: l1 ++ a) \* \[res = vint (x + sum (to_int_list l1))] ⟹
x0 ⇝ vlist ((vint x :: l1) ++ a) \* \[res = vint (sum (to_int_list (vint x :: l1)))]
```

What we would like

```
res = vint (x + sum (to_int_list l1)) → res = vint (sum (to_int_list (vint x :: l1)))
```

What we would like

```
No more goals
```

The mechanisation at a glance

- Other things formalised:
  - Programs, big-semantics
  - ::: (pairs), (history) triples
  - Soundness
- 4700 LoC, on top of [Charguéraud 20]

Takeaways

https://github.com/dariusf/staged

- An alternative program logic formulation
  - New primitives; no wands, weakest preconditions, or step-indexing
  - Higher-order + effects
- Other views
  - Refinement between abstract programs
  - Triples with syntactic reasoning
  - Manipulating verification conditions directly
- Future work
  - Automation to support certification
  - Other applications of staged logic [Song 24]

Comparison with CFML

- A characteristic formula is a relation between precondition and postcondition, i.e. `cf : expr -> (assertion -> assertion -> Prop)`

- A staged formula is a syntactic entity whose semantics relates pre- and post-*states*
  - This allows more kinds of syntactic reasoning, e.g. mentioning unknown functions

$$\mathcal{E}, h_1, h_2, v \vDash \varphi$$

Biabduction

$H_A * H_1 \vdash H_2 * H_F$ [Calcagno 09]

Deeply embedded, for induction over derivations

```
Inductive biab : hprop -> hprop -> hprop -> hprop -> Prop :=
| b_base_empty : forall Hf,
  biab \[] Hf \[] Hf

| b_pts_match : forall a b H1 H2 Ha Hf x,
  biab Ha H1 H2 Hf ->
  biab (\[a=b] \* Ha) (x~~>a \* H1) (x~~>b \* H2) Hf
...
```

```
Lemma biab_sound : forall Ha H1 H2 Hf,
  biab Ha H1 H2 Hf ->
  Ha \* H1 ==> H2 \* Hf.
```

Why mechanise an automated verifier?

- Check claims in paper
- Clarify ideas: find the simplest version of each concept
- Communicate: other people can try the logic and build intuition
- Certification: validate implementation, not just ideas
- Support new work: work manually on harder proofs, broaden fragment that can be fully automated