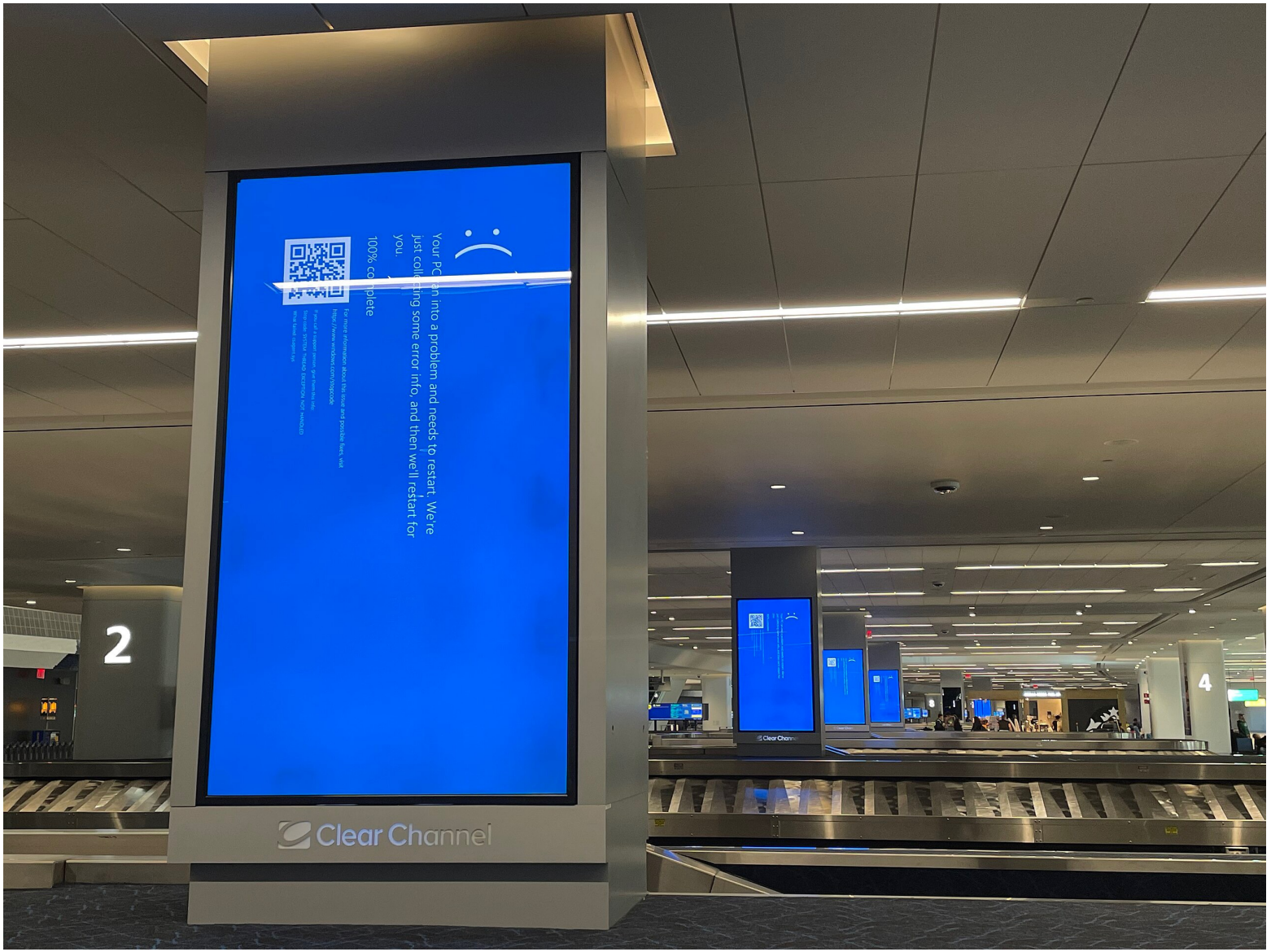
Automated Verification of Effectful Higher-order Programs

Thesis Proposal

Darius Foo


2
Clear Channel
4

BBC

Home News Sport Business Innovation Culture Arts Travel Earth Audio Video Live

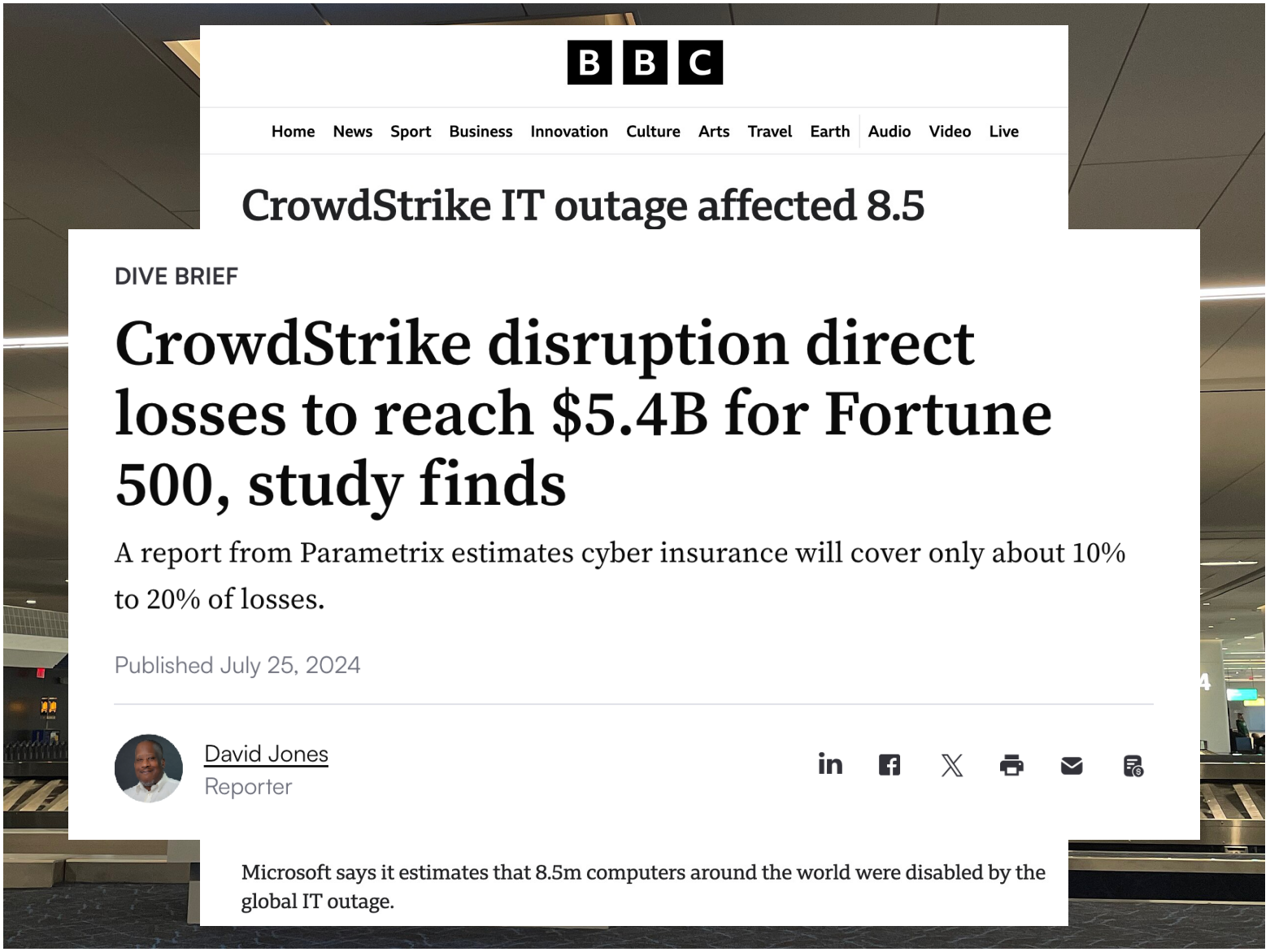
CrowdStrike IT outage affected 8.5 million Windows devices, Microsoft says

21 July 2024

Share Save

Joe Tidy Cyber correspondent, BBC News

Microsoft says it estimates that 8.5m computers around the world were disabled by the global IT outage.



BBC

Home News Sport Business Innovation Culture Arts Travel Earth Audio Video Live

CrowdStrike IT outage affected 8.5

DIVE BRIEF

CrowdStrike disruption direct losses to reach $5.4B for Fortune 500, study finds

A report from Parametrix estimates cyber insurance will cover only about 10% to 20% of losses.

Published July 25, 2024

David Jones
Reporter

Microsoft says it estimates that 8.5m computers around the world were disabled by the global IT outage.

Bugs can be *very* expensive

How do we ensure that a program is free of bugs?

The gold standard: *verify* it using an appropriate *program logic*.

Program verification scales to large projects…

COMPCERT

COMPILERS YOU CAN *FORMALLY* TRUST

ACM Software System Award 2021

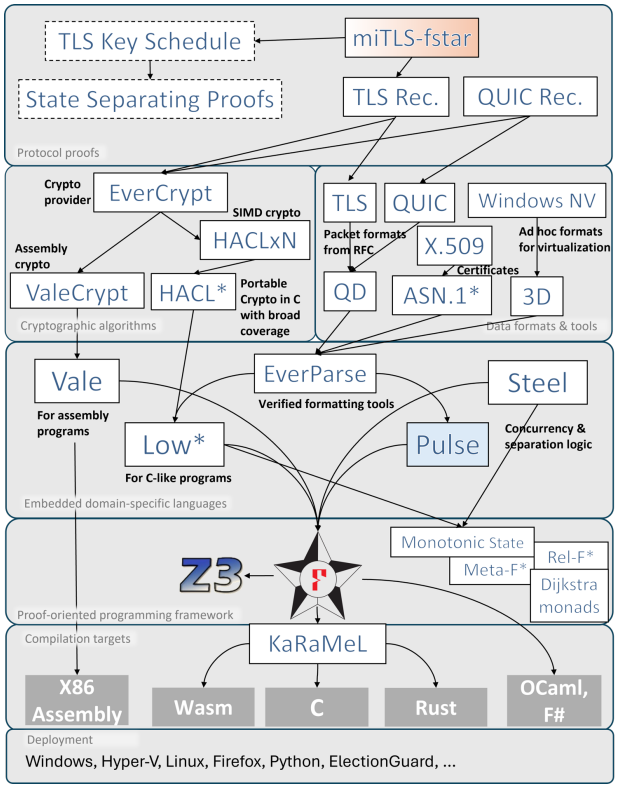
ACM SIGPLAN Programming Languages Software Award 2022

The CompCert project investigates the formal verification of realistic compilers usable for critical embedded software. Such verified compilers come with a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program. By ruling out the possibility of compiler-introduced bugs, verified compilers strengthen the guarantees that can be obtained by applying formal methods to source programs.

The main result of the project is the CompCert C verified compiler, a high-assurance compiler for almost all of the C language (ISO C 2011), generating efficient code for the ARM, PowerPC, RISC-V and x86 processors.

MENU

- Home
- Partners
- Motivations
- Research
- The Compcert C compiler
- Downloads
- Publications



… but remains rather costly

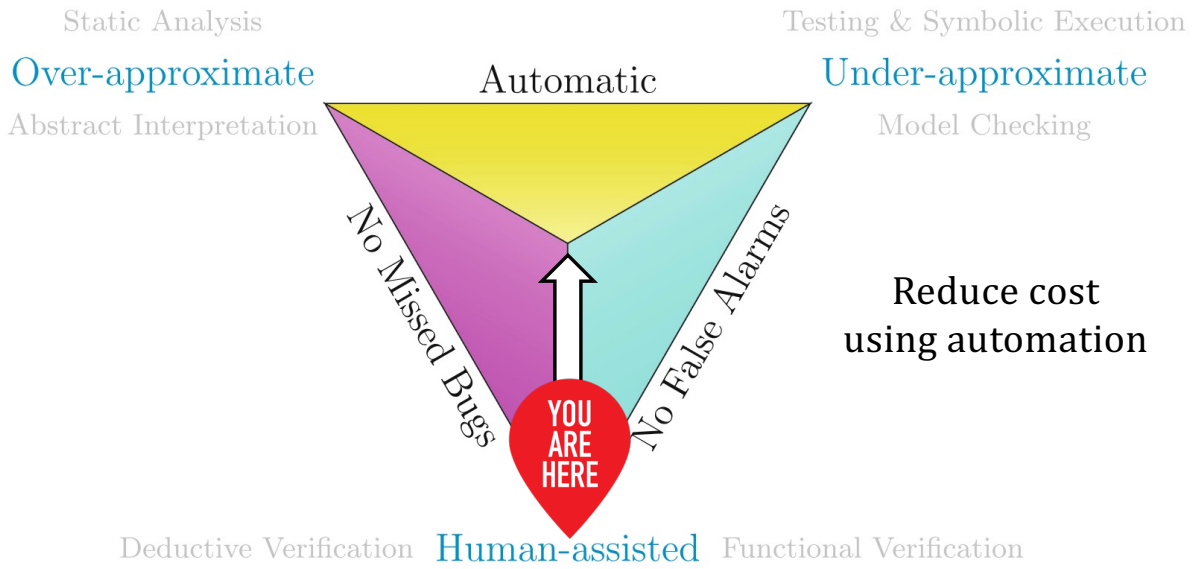


Fig. 2. The software verification pyramid with the six schools.

"A pyramid of (formal) software verification", Martin Brain and Elizabeth Polgreen, FM24

Bugs can be *very* expensive

How do we ensure that a program is free of bugs?

The gold standard: *verify* it using an appropriate *program logic*.

Enabling automation is the main means of reducing the cost of verification.

Effectful higher-order programs

It is very common in modern languages to mix effects and higher-order functions.

Yet, automated reasoning about this combination of features remains challenging.

The challenges stem from the need to produce auxiliary specifications upfront.

Problem

Auxiliary specifications require insight and commit to an abstraction immediately.

It is not easy to infer or synthesise these abstractions, and they hamper automation.

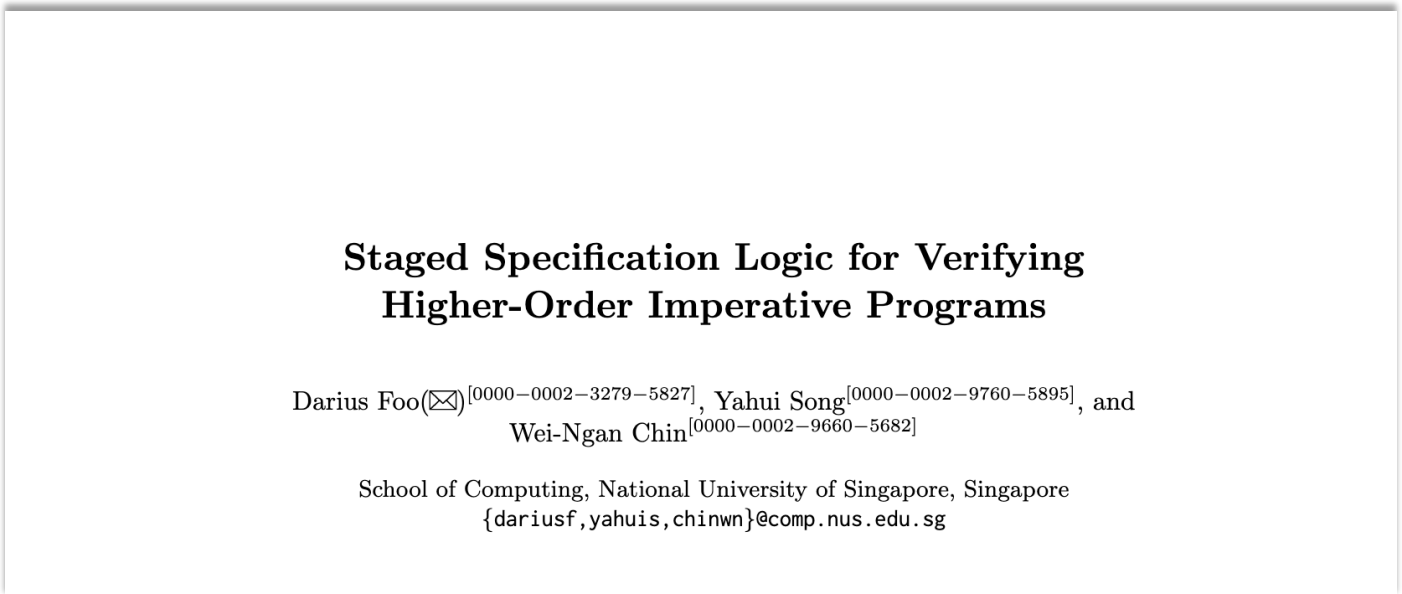
Towards a solution

Insight: *deferring abstraction* is a way around the need for intermediate specifications.

If we enrich specifications with the ability to describe effects precisely, we can afford to wait until more information is known before we are forced to abstract.

Staged logic

A simple program logic for automated verification of 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

Staged logic

A simple program logic for automated verification of effectful higher-order programs.

It provides the means of deferring abstraction using precise specifications of effects.

This simplifies both specifications and proofs, and applies to diverse kinds of effects.

Thesis statement

Staged logic is an effective and sound means of automating the verification of effectful higher-order programs.

Outline

- Reasoning about imperative higher-order programs
- Reasoning about effect handlers
- Reasoning about delimited continuations
- Mechanising staged logic
- Conclusion

Outline

- Reasoning about imperative higher-order programs
- Reasoning about effect handlers
- Reasoning about delimited continuations
- Mechanising staged logic
- Conclusion

An imperative 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
```

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

How do we prove it automatically?

The traditional approach

- An intermediate specification for *foldr*

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

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

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

The traditional approach

- An intermediate specification for *foldr*

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

The traditional approach

- An intermediate specification for *foldr*

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

The traditional approach

- Parameterise specification of *foldr* over properties/invariants

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) predicate relating suffix of input list traversed to result

The invariant can mention *resources*, such as the mutable reference *f* captures, giving us a way to talk about its effects

The traditional approach

- Parameterise specification of *foldr* over properties/invariants
  - i.e. come up with an adequate abstraction of its behaviour
- Automation is difficult
  - How to infer invariants to be supplied at call sites?
  - How to infer abstraction? Clients require different parameterisations

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.

Iris Lecture Notes, Sec. 5

The traditional approach

- Parameterise specification of *foldr* over properties/invariants
  - i.e. come up with an adequate abstraction of its behaviour
- Automation is difficult
  - Automated verifiers (Cameleer, Why3, …) don't handle such programs

How do we solve this problem?

Rethinking the means of specification

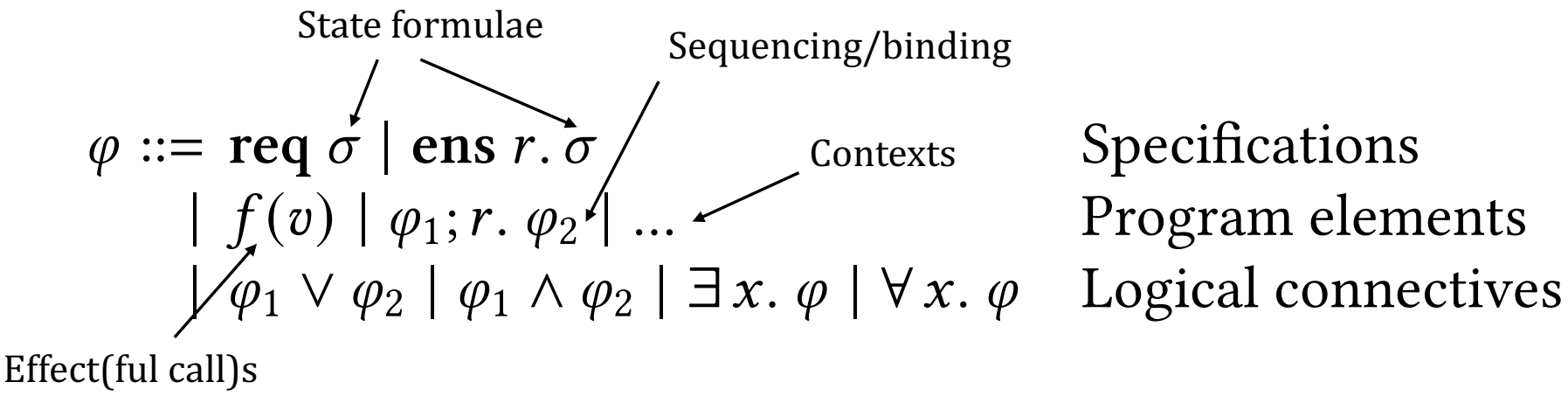
Problem: our underlying (separation) logic cannot talk about effects easily – it took insight to use the invariant on resources as a proxy for *foldr*'s effects.

We would like a logic where we can

(1) model the essential parts of *foldr* (the *ordering* of effects) precisely, while

(2) eliding nonessential detail.

Staged logic



- A logic of specifications, enriched with syntax to describe effects and their contexts
- A higher-order *verification intermediate language* (e.g. Viper), with specifications embedded in programs

Compared to Hoare logic

Staged logic generalises* Hoare logic.

ens has a dual meaning in the history

$$\{\, P \,\}\, e\, \{\, r.\, Q \,\} \triangleq \{\, \mathbf{ens}\ P \,\}\, e\, \{\, \mathbf{ens}\ r.\, Q \,\}$$

History

$$\boxed{\{\, \varphi \,\}\, e\, \{\, \varphi \,\}}$$

$$\{\, P \,\}\, e\, \{\, r.\, Q \,\} \triangleq e ::: \mathbf{req}\ P; \mathbf{ens}\ r.\, Q$$

$$\boxed{e ::: \varphi}$$

where $e ::: \varphi \triangleq \{\, \mathbf{ens}\ emp \,\}\, e\, \{\, \varphi \,\}$

Any Hoare logic derivation has a staged logic equivalent.

Compared to Hoare logic

Programs containing higher-order unknowns (or, more generally, uninterpreted effects) don't have a clear "best" specification in Hoare logic.

Staged logic	**Hoare logic**
$$\frac{\{\,P\,\}\,e\,\{\,r.\;Q\,\}}{e; f(a) \;::\; \mathbf{req}\; P; \mathbf{ens}\; r.\; Q; f(a)}$$ $$f(g(x)) \;::\; g(x); r.\; f(r)$$	Quantify over invariants, pre/post? How to parameterise properties?

foldr

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

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

The specification

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

The proof, very briefly

By induction on xs

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

The proof, very briefly

Recursive case

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

The proof, very briefly

Rewrite using the IH

$$t \leq xs$$

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

The proof, very briefly

Rewrite using the IH

$$\begin{array}{l}
\mathbf{ens}\ (xs = h :: t); \\
\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ r.\ (x \mapsto (t +\!\!+ a_1) * [r = sum\ t]); r. \\
g(h, r) \\
\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}\ r.\ x \mapsto (xs +\!\!+ a) * [r = sum\ xs]
\end{array}$$

The proof, very briefly

Unfold the closure

$$
\begin{array}{l}
\mathbf{ens}\ (xs = h :: t); \\
\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ r.\ (x \mapsto (t +\!\!+ a_1) * [r = sum\ t]); r. \\
g(h, r) \\
\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}\ r.\ x \mapsto (xs +\!\!+ a) * [r = sum\ xs]
\end{array}
$$

The proof, very briefly

Unfold the closure

$$
\begin{aligned}
&\mathbf{ens}\ (xs = h :: t);\\
&\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ r.\ (x \mapsto (t +\!\!+ a_1) * [r = sum\ t]); r.\\
&\forall z, \mathbf{req}\ x \mapsto z; \mathbf{ens}\ r_1.\ x \mapsto (h :: z) * [r_1 = h + r]\\
&\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}\ r.\ x \mapsto (xs +\!\!+ a) * [r = sum\ xs]
\end{aligned}
$$

The proof, very briefly

Normalise

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

$$\mathbf{ens}\ (xs = h :: t);$$
$$\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ r.\ (x \mapsto (t +\!\!+ a_1) * [r = sum\ t]); r.$$
$$\forall z, \mathbf{req}\ x \mapsto z; \mathbf{ens}\ r_1.\ x \mapsto (h :: z) * [r_1 = h + r]$$
$$\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}\ r.\ x \mapsto (xs +\!\!+ a) * [r = sum\ xs]$$

The proof, very briefly

Normalise

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

$$\begin{aligned}
&\mathbf{ens}\ (xs = h :: t);\\
&\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ r.\ (x \mapsto (t \mathbin{+\!\!+} a_1) * [r = sum\ t]); r.\\
&\forall z, \mathbf{req}\ x \mapsto z; \mathbf{ens}\ r_1.\ x \mapsto (h :: z) * [r_1 = h + r]\\
&\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}\ r.\ x \mapsto (xs \mathbin{+\!\!+} a) * [r = sum\ xs]
\end{aligned}$$

The proof, very briefly

Normalise

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

$$\begin{array}{l}
\mathbf{ens}\ (xs = h :: t); \\
\forall a_1, \mathbf{req}\ x \mapsto a_1; \mathbf{ens}\ r.\ (x \mapsto (t \mathbin{+\!\!+} a_1) * [r = sum\ t]); r. \\
\forall z, \mathbf{req}\ x \mapsto z; \mathbf{ens}\ r_1.\ x \mapsto (h :: z) * [r_1 = h + r] \\
\sqsubseteq \mathbf{req}\ x \mapsto a; \mathbf{ens}\ r.\ x \mapsto (xs \mathbin{+\!\!+} a) * [r = sum\ xs]
\end{array}$$

The proof, very briefly

Normalise

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

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

The proof, very briefly

Contravariance of req

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

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

The proof, very briefly

Contravariance of req

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

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

The proof, very briefly

Contravariance of req

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

The proof, very briefly

Covariance of ens

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

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

The proof, very briefly

Separation logic entailment

$$x \mapsto (h :: (t \mathbin{+\!\!+} a)) * \\ \qquad [res = h + sum\ t \wedge xs = h :: t] \\ \vdash x \mapsto (xs \mathbin{+\!\!+} a) * [res = sum\ xs]$$

The proof, very briefly

SMT (and some properties of append and cons)

$$res = h + (sum\ t) \wedge xs = h :: t$$
$$\Longrightarrow h :: (t +\!\!+ a) = xs +\!\!+ a \wedge res = sum\ xs$$

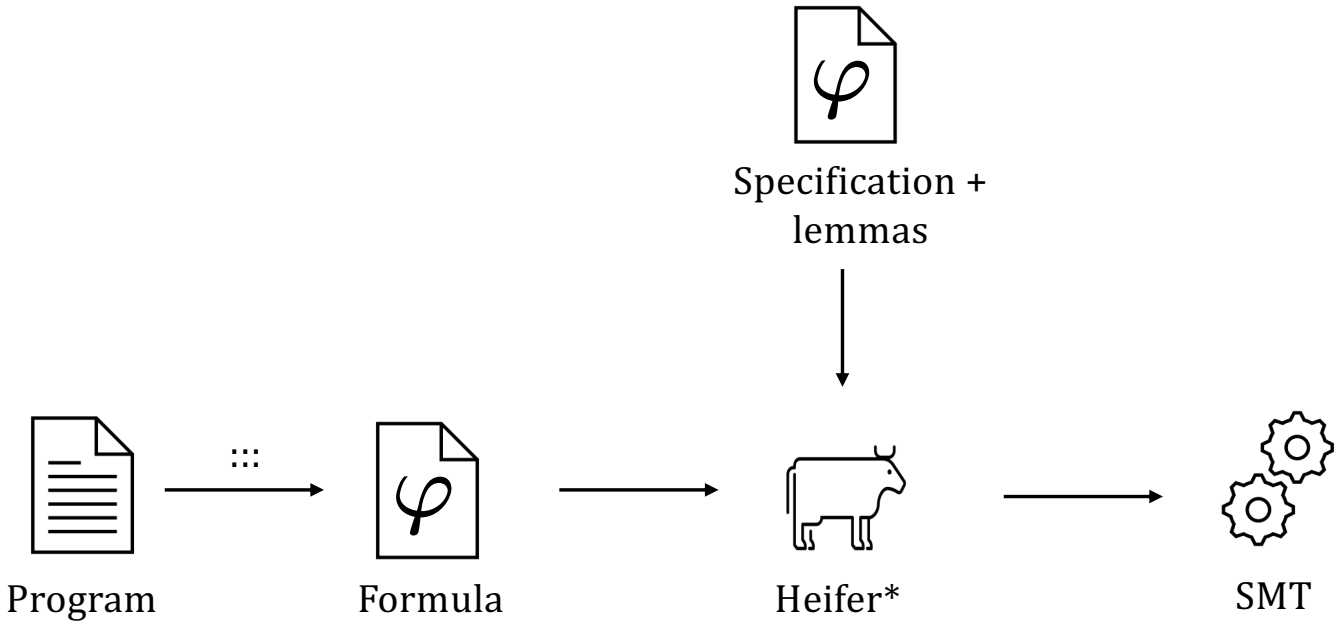
Key ideas

"Inequational" reasoning, with *abstractions* of programs.

Biabduction as a means of symbolic execution.

Invariants still appear in the induction hypothesis, but inferring one is less difficult in practice than arbitrary higher-order parameters.

An automated verifier for staged logic



*Higher-order Effectful Imperative Function Entailments & Reasoning

Evaluation

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 "-" denoting incomparable. We use T to denote the total verification time (in seconds) and T_P to record the time spent on the external provers.)

	Heifer				Cameleer [24]			Prusti [33]		
Benchmark	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 [29]	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 [13]	14	6	0.31	0.28		✗		13	9	6.24
counter	16	4	0.24	0.18		✗		11	7	6.37
lambda	13	5	0.25	0.22		✗			-	
	197	73			45	112		37	27	

Conclusion

Deferring abstraction allowed us to avoid encoding effects as invariants on resources.

Outline

- Reasoning about imperative higher-order programs
- Reasoning about effect handlers
- Reasoning about delimited continuations
- Mechanising staged logic
- Conclusion

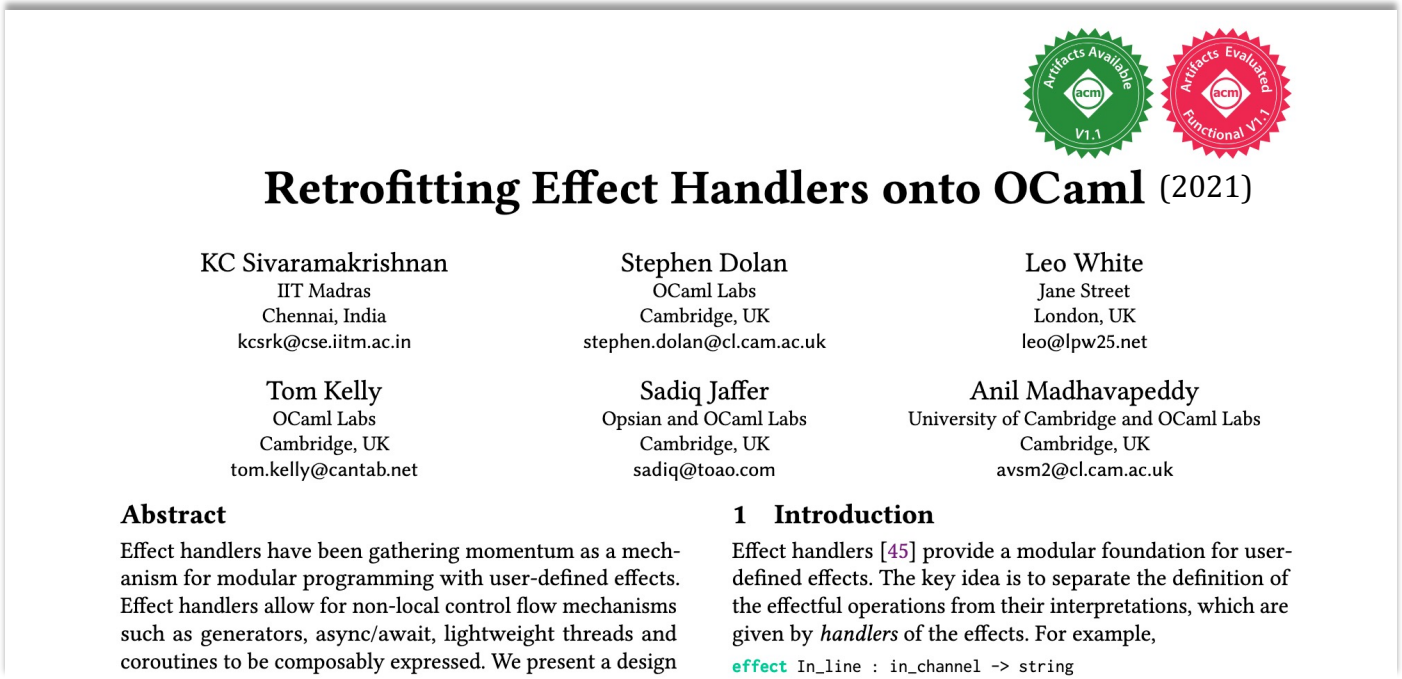
Effect handlers

A language feature for implementing non-local control flow.

- Exceptions
- Lightweight threads
- Coroutines
- Generators
- Automatic differentiation
- Nondeterminism
- …

Effect handlers

OCaml 5 supports them natively and uses them as the main structuring mechanism for concurrency.



Retrofitting Effect Handlers onto OCaml (2021)

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@toao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

Abstract

Effect handlers have been gathering momentum as a mechanism for modular programming with user-defined effects. Effect handlers allow for non-local control flow mechanisms such as generators, async/await, lightweight threads and coroutines to be composably expressed. We present a design

1 Introduction

Effect handlers [45] provide a modular foundation for user-defined effects. The key idea is to separate the definition of the effectful operations from their interpretations, which are given by *handlers* of the effects. For example,

```
effect In_line : in_channel -> string
```

Effect handlers

OCaml 5 supports them natively and uses them as the main structuring mechanism for concurrency.

How do we reason about programs which use effect handlers?

Exchange

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

exchange v

$$\{res.\ x \mapsto v * [res = a]\}$$

```
let client () =
  exchange (exchange 42)
```

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

client ()

$$\{res.\ x \mapsto a * [res = 42]\}$$

Exchange using an effect handler

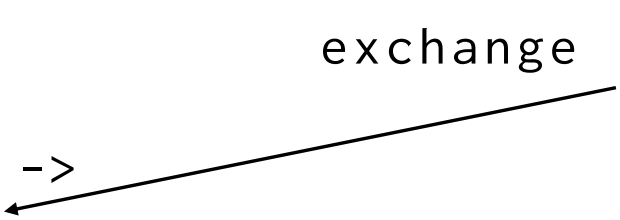
```
type _ Effect.t += XCHG: int -> int Effect.t

let exchange n = perform (XCHG n)

let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
    let old_p = !p in
    p := n;
    continue k old_p
```

```
let client () =
  exchange (exchange 42)
```

Exchange using an effect handler

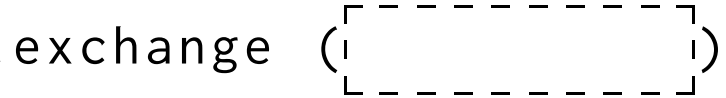
```
let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
    let old_p = !p in
    p := n;
    continue k old_p
```

```
let client () =
  exchange (exchange 42)
```



Exchange using an effect handler

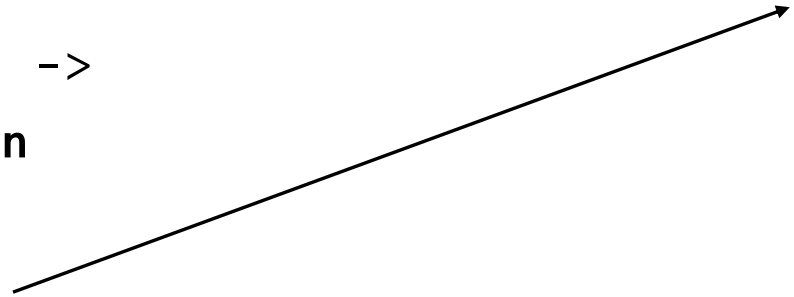
```
let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
  let old_p = !p in
  p := n;
  continue k old_p
```



Exchange using an effect handler

```
let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
    let old_p = !p in
    p := n;
    continue k old_p
```

```
let client () =
  exchange (   old_p   )
```



Exchange using an effect handler

```
let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
    let old_p = !p in
    p := n;
    continue k old_p
```

```
let client () =
  exchange (   old_p   )
```

The handler is *deep*: it remains installed, *recursively* handling the next exchange

Exchange using an effect handler

```
let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
    let old_p = !p in
    p := n;
    continue k old_p
```

```
let client () =
  exchange (exchange 42)
```

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

`main ()`

$$\{res.\ x \mapsto a * [res = 42]\}$$

Verifying programs with effect handlers

How do we prove such a triple?

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

```
main ()
```

$$\{res.\ x \mapsto a * [res = 42]\}$$

Verifying programs with effect handlers

The state of the art today: two auxiliary specifications are required

1. A *protocol*, a pre/post specification for an effect

```
protocol XCHG x:
  ensures !p = x && reply = old !p
  modifies p
```

2. A *handler invariant*, re-established at the end of a handler body

```
try client () with ...
(*@ try_ensures !p = old !p
      && result = 42 *)
```

A Separation Logic for Effect Handlers, de Vilhena et al., POPL 21
A Framework for the Automated Verification of Algebraic Effects and Handlers, Soares et al. arXiv 23

Verifying clients (SOTA)

```
let client () =
  exchange (exchange 42)
```

```
protocol XCHG x:
  ensures !p = x && reply = old !p
  modifies p
```

```
let exchange n = perform (XCHG n)
(*@ ensures !p = n && result = old !p
    performs XCHG *)
```

Verifying handlers (SOTA)

The specifications of (most of) the handler and perform are *dual*

```
let main () =
  let p = ref 0 in
  try client () with
  | effect XCHG n, k ->
    let old_p = !p in
    p := n;
    continue k old_p
```

```
protocol XCHG x:
  ensures !p = x && reply = old !p
  modifies p
```

Assume precondition

Prove postcondition

The handler invariant describes its recursive behaviour

Assume handler invariant

Re-establish handler invariant

```
try client () with ...
(*@ try_ensures !p = old !p
      && result = 42 *)
```

Verifying effect handlers with protocols

An elegant, modular approach for Hoare logic. However…

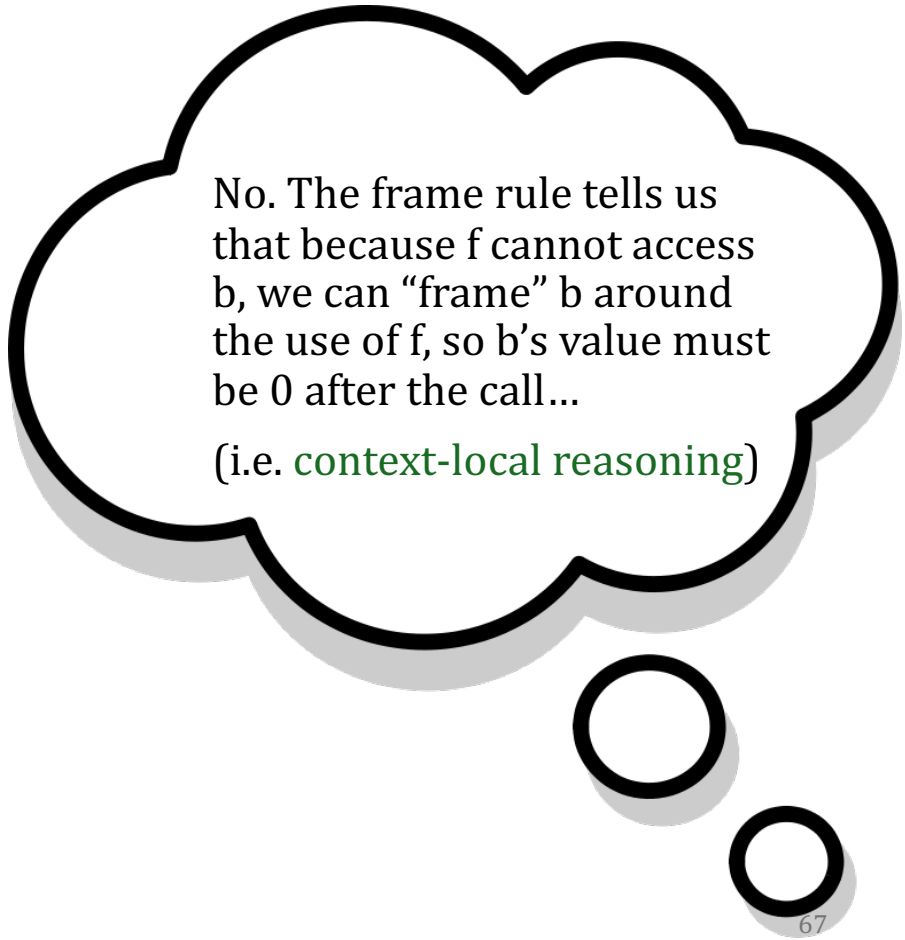
Problem #1 an abstraction of the handler is needed upfront to verify a client; effects are always interpreted

Problem #2 the frame rule does not generally hold in the presence of multishot continuations

Problem #2: multishot continuations

Is it possible for the assertion to fail?

```
let call f =
  let b = ref 0 in
  f ();
  b := !b + 1;
  assert (!b = 1)
```



Problem #2: multishot continuations

Is it possible for the assertion to fail? Yes!

```
let call f =
  let b = ref 0 in
  f ();
  b := !b + 1;
  assert (!b = 1)
```

```
let main () =
  try
    call (fun () -> perform E)
  with
  | effect E, k ->
    continue k ();
    continue k () ❌
```

The problem is that we can manipulate contexts, which can capture resources…

Problem #2: multishot continuations

In the SOTA approach today, for soundness, when a perform occurs, resources *disappear* and cannot be framed around uses of effects.

Context-local reasoning using protocols is limited with multishot continuations.

Proof of Programs with Effect Handlers, de Vilhena, PhD thesis 22

Towards a solution

Idea: to reason context-locally about effects, give effects a first-class treatment in the logic.

$$\{\ \mathbf{ens}\ \mathit{emp}\ \}\ \mathit{perform}\ E()\ \{\ E()\ \}$$

Staged logic naturally supports this idea, as it allows descriptions of effects, as well as their *contexts*.

$$\{\ \mathbf{ens}\ \mathit{emp}\ \}\ \mathit{perform}\ E(); f()\ \{\ E(); f()\ \}$$

Context-local descriptions of effects

This idea goes quite far. We can use it to reason naturally about multishot continuations *and* resources.

$$\mathbf{assert}\ D \triangleq \mathbf{req}\ D; \mathbf{ens}\ r.\ D \wedge r{=}()$$

$$\mathit{call} \triangleq \forall\, b.\ \mathbf{ens}\ b \mapsto 0; E; \forall\, v.\ \mathbf{req}\ b \mapsto v; \mathbf{ens}\ b \mapsto v{+}1; \mathbf{assert}\ b \mapsto 1$$

$$\mathbf{try}\ \mathit{call}\ \mathbf{with}\ E, k \to -1 \sqsubseteq \forall\, b.\ \mathbf{ens}\ r.\ b \mapsto 0 \wedge r{=}-1$$

$$\mathbf{try}\ \mathit{call}\ \mathbf{with}\ E, k \to k() \sqsubseteq \forall\, b.\ \mathbf{ens}\ r.\ b \mapsto 1 \wedge r{=}()$$

$$\mathbf{try}\ \mathit{call}\ \mathbf{with}\ E, k \to k(); k() \sqsubseteq \mathbf{req}\ \mathit{false}$$

Context-local descriptions of effects

Symbolic execution does a lot of work in the proofs.

$$
\begin{array}{lll}
 & \textbf{try}\ \mathit{call}\ \textbf{with}\ E, k \to k() \sqsubseteq \forall\, b.\ \textbf{ens}\ r.\ b \mapsto 1 \wedge r{=}() & \\
\sqsubseteq & \textbf{try}\ (\forall\, b.\ \textbf{ens}\ b \mapsto 0; E; \ldots)\ \textbf{with}\ E, k \to k() & \text{Unfold} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 0; \textbf{try}\ (E; \ldots)\ \textbf{with}\ E, k \to k() & \text{Float} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 0; \forall\, v.\ \textbf{req}\ b \mapsto v; \textbf{ens}\ b \mapsto v{+}1; \textbf{assert}\ b \mapsto 1 & \text{Handle} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 0{+}1; \textbf{assert}\ b \mapsto 1 & \text{Biab} \\
\sqsubseteq & \forall\, b.\ \textbf{ens}\ b \mapsto 1 \wedge r{=}0 & \text{Biab}
\end{array}
$$

Exchange using staged logic

$$
\begin{aligned}
\mathit{client} \triangleq\ & \mathit{XCHG}(42); r.\ \mathit{XCHG}(r) \\
\mathit{main} \triangleq\ & \forall\, p.\ \mathbf{ens}\ p \mapsto 0; \mathbf{try}\ \mathit{client}\ \mathbf{with}\ \mathit{XCHG}(n), k \to \\
& \quad (\forall\, o.\ \mathbf{req}\ p \mapsto o; \mathbf{ens}\ p \mapsto n; k(o))
\end{aligned}
$$

$$
\begin{array}{lll}
 & \mathbf{ens}\ p \mapsto 0; \mathbf{try}\ \mathit{client}\ \mathbf{with}\ \mathit{XCHG}(n), k \to \ldots & \\
\sqsubseteq & \mathbf{ens}\ p \mapsto 0; & \text{Handle} \\
 & \mathbf{ens}\ k{=}(\lambda x.\ \mathbf{try}\ \mathit{XCHG}(x)\ \mathbf{with}\ \mathit{XCHG}(n), k \to \ldots) & \\
 & \forall\, o.\ \mathbf{req}\ p \mapsto o; \mathbf{ens}\ p \mapsto 42; k(o) & \\
\sqsubseteq & \mathbf{ens}\ k{=}(\lambda x.\ \mathbf{try}\ \mathit{XCHG}(x)\ \mathbf{with}\ \mathit{XCHG}(n), k \to \ldots); & \text{Biab} \\
 & \mathbf{ens}\ p \mapsto 42; k(0) & \\
\sqsubseteq & \mathbf{ens}\ p \mapsto 42; \mathbf{try}\ \mathit{XCHG}(0)\ \mathbf{with}\ \mathit{XCHG}(n), k \to \ldots & \text{Simplify} \\
\sqsubseteq & \mathbf{ens}\ p \mapsto 42; \forall\, o.\ \mathbf{req}\ p \mapsto o; \mathbf{ens}\ p \mapsto n; k(o) & \text{Handle} \\
\sqsubseteq & \mathbf{ens}\ r.\ p \mapsto 0 \wedge r{=}42 & \text{Biab}
\end{array}
$$

Verifying effect handlers with staged logic

Solution #1 assume effects are uninterpreted, only committing to an interpretation given a handler; no need for auxiliary specifications

Solution #2 use the frame rule only to describe states, not effects; multishot continuations and resources can coexist easily

Evaluation

Table 1. Experimental Results.

#	Program	Ind	MultiS	ImpureC	HO	LoC	LoS	Total(s)	AskZ3(s)
1	State monad	✗	✗	✓	✗	126	16	8.54	6.21
2	Inductive sum	✓	✗	✓	✗	41	11	1.68	1.28
3	Flip-N (Deep Right Rec) (Fig. 7)	✓	✓	✓	✗	39	10	2.09	1.52
4	Flip-N (Deep Left Rec)	✓	✓	✓	✗	45	13	2.03	1.53
5	Flip-N (Shallow Right Rec)	✗	✓	✓	✗	37	11	5.08	3.18
6	Flip-N (Shallow Left Rec)	✓	✓	✓	✗	64	23	6.75	4.26
7	McCarthy's amb operator (Fig. 25)	✓	✓	✓	✓	109	45	7.71	5.34
	Total	-	-	-	-	461	129	33.88	23.32

Conclusion

Deferring abstraction allowed us to avoid encoding handlers as a protocol and handler invariant.

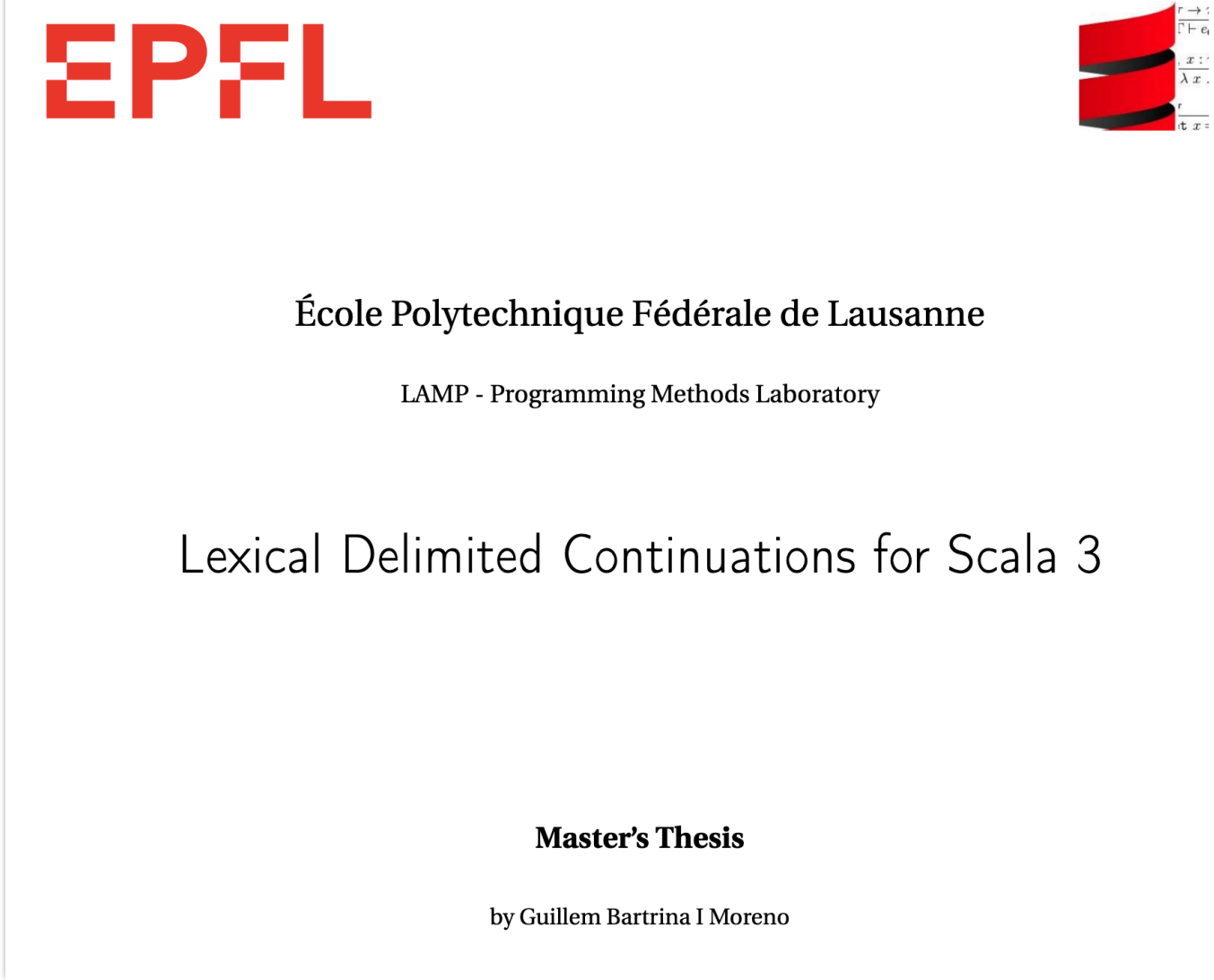
Outline

- Reasoning about imperative higher-order programs
- Reasoning about effect handlers
- Reasoning about delimited continuations
- Mechanising staged logic
- Conclusion

Delimited continuations

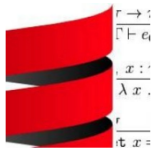
Effect handlers represent only one set of design choices. Other control operators have been proposed for implementing effects.

Moreover, these operators are making their way into mainstream languages.



- shift_0 in Scala 3 (2025)

EPFL



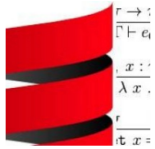
Project Loom: Fibers and Continuations for the Java Virtual Machine

Overview

Project Loom's mission is to make it easier to write, debug, profile and maintain concurrent applications meeting today's requirements. Threads, provided by Java from its first day, are a natural and convenient concurrency construct (putting aside the separate question of communication among threads) which is being supplanted by less convenient abstractions because their current implementation as OS kernel threads is insufficient for meeting modern demands, and wasteful in computing resources that are particularly valuable in the cloud. Project Loom will introduce fibers as lightweight, efficient threads managed by the Java Virtual Machine, that let developers use the same simple abstraction but with better performance and lower footprint. We want to make concurrency simple(r) again! A fiber is made of two components — a continuation and a scheduler. As Java already has an excellent scheduler in the form of `ForkJoinPool`, fibers will be implemented by adding continuations to the JVM.

- shift_0 in Scala 3 (2025)
- shift_0 in Java 21 (2023)

EPFL



- shift_0 in Scala 3 (2025)
- shift_0 in Java 21 (2023)
- control_0 in GHC (2022)

Project Loom: Fibers and Continuations for the Java Virtual Machine

Overview

Project Loom's mission is to make it easier to
meeting today's requirements. Threads, provi
concurrency construct (putting aside the sepa
supplanted by less convenient abstractions b
insufficient for meeting modern demands, and
the cloud. Project Loom will introduce fibers a
Machine, that let developers use the same sir
We want to make concurrency simple(r) again
scheduler. As Java already has an excellent s
by adding continuations to the JVM.

Glasgow Haskell Compiler / GHC / Merge requests / !7942

Native, first-class, delimited continuations

Code

Merged **Alexis King** requested to merge `lexi.lambda/ghc:first-cl…` into `master` 3 years ago

Overview 64 | Commits 1 | Pipelines 24 | Changes 56 | 4 unresolved threads

This MR implements GHC proposal 313: Delimited continuation primops by adding native support for delimited continuations to the GHC RTS.

All things considered, the patch is relatively small. It almost exclusively consists of changes to the RTS; the compiler itself is essentially unaffected. The primops come with fairly extensive Haddock documentation, which can be found in `primops.txt.pp`. For an overview of the implementation strategy, as well as a discussion of several of the finer details, see the Notes in `rts/Continuation.c`.

Assignee
Marge Bot

Reviewer
Approval is optional
Ben Gamari

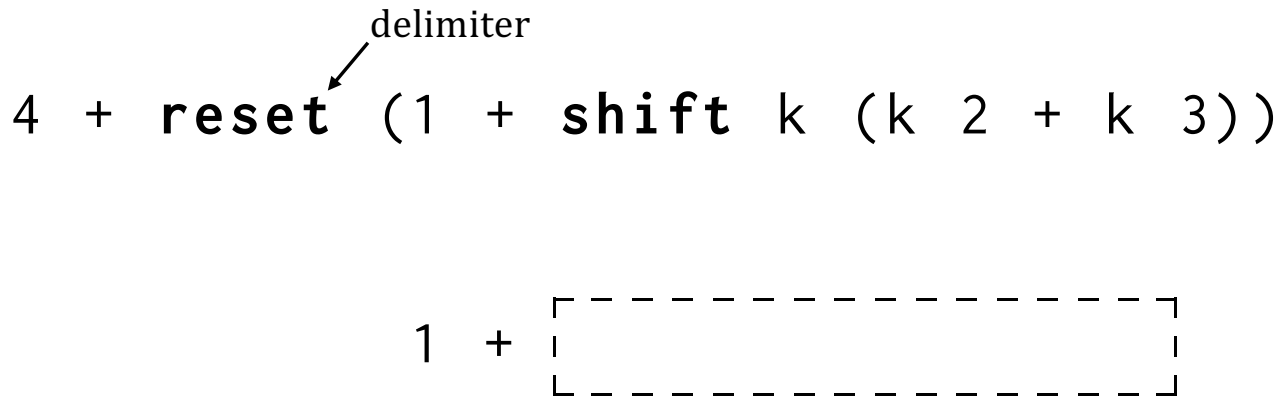
Delimited continuations

Effect handlers represent only one set of design choices. Other control operators have been proposed for implementing effects.

Moreover, these operators are making their way into mainstream languages.

We would like to generalise staged logic to handle them.

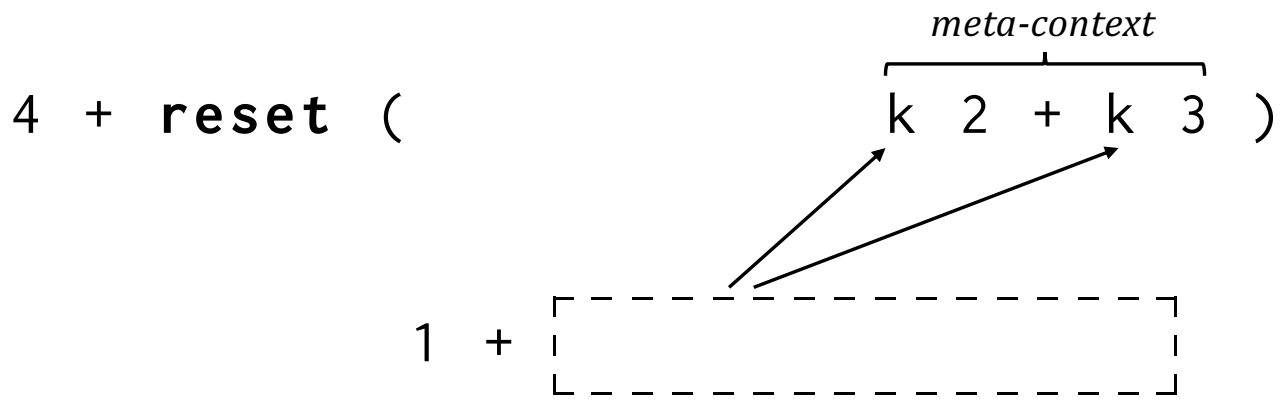
shift and reset



$$\langle v \rangle \Longrightarrow v$$

$$\langle E[\textbf{shift}\ k.\ e] \rangle \Longrightarrow \langle (\lambda k.\ e)\ (\lambda v.\ \langle E[v] \rangle) \rangle$$

shift and reset



$$\langle v \rangle \Longrightarrow v$$

$$\langle E[\textbf{shift}\ k.\ e] \rangle \Longrightarrow \langle (\lambda k.\ e)\ (\lambda v.\ \langle E[v] \rangle) \rangle$$

shift and reset

$$4 + \texttt{reset}\ (\quad (1 + \boxed{2}) + (1 + \boxed{3})\quad) = 11$$

$$\langle v \rangle \Longrightarrow v$$
$$\langle E[\textbf{shift}\ k.\ e] \rangle \Longrightarrow \langle (\lambda k.\ e)\ (\lambda v.\ \langle E[v] \rangle) \rangle$$

shift and reset, and effect handlers

In an untyped setting, shift and reset can be macro-expressed as effect handlers.

$$\mathbf{shift}_0\ k.\ e_b \approx \mathbf{perform}\ \mathit{Shift}$$

$$\langle e \rangle \approx \mathbf{try}\ e\ \mathbf{with}\ \mathit{Shift}, k \longrightarrow e_b$$

However, they encourage different styles of reasoning (and hence different reasoning principles).

On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control, Forster et al. ICFP 17

Context-local reasoning

Context-local reasoning is similarly difficult.

The SOTA for verification uses protocols, or *non-context-local triples*: if e contains a shift, it may manipulate the context C.

$$\frac{}{\{\, P \,\}\, C[e]\, \{\, Q \,\}}$$

This falls back to essentially global reasoning.

Context-local descriptions of shifts

We can use the same solution: model a shift in staged logic directly, uninterpreted.

$$\frac{e ::: \varphi}{\{\, \mathbf{ens}\ emp \,\}\ \mathbf{shift}\ k.\ e\ \{\, \mathbf{shift}\ k.\ \varphi \,\}}$$

The crux of this: defining the semantics of a shift *outside* a reset (aka the "bubble-up semantics"), which includes a *partial* continuation.

$$\mathbf{shift}\ k.\ e; f() ::: \mathbf{shift}\ k.\ \varphi; f()$$

Other operators

Given a way to state the semantics of shift, it is natural to generalise the framework to the other operators.

$$
\begin{aligned}
\langle v \rangle &\Longrightarrow v \\
\langle E[\mathbf{shift}\ k.\ e] \rangle &\Longrightarrow \langle (\lambda k.\ e)\ (\lambda v.\ \langle E[v] \rangle) \rangle \\
\langle E[\mathbf{shift}_0\ k.\ e] \rangle &\Longrightarrow (\lambda k.\ e)\ (\lambda v.\ \langle E[v] \rangle) \\
\langle E[\mathbf{control}\ k.\ e] \rangle &\Longrightarrow \langle (\lambda k.\ e)\ (\lambda v.\ E[v]) \rangle \\
\langle E[\mathbf{control}_0\ k.\ e] \rangle &\Longrightarrow (\lambda k.\ e)\ (\lambda v.\ E[v])
\end{aligned}
$$

A Monadic Framework for Delimited Continuations, Dybvig et al. JFP 05

Traverse/reverse with control and shift

```
let visit xs =
  match xs with
  | [] -> []
  | x :: xs1 ->
    visit (shift k (x :: k xs))

let traverse xs = reset (visit xs)
```

```
let visit' xs =
  match xs with
  | [] -> []
  | x :: xs1 ->
    visit' (control k (x :: k xs))

let reverse xs = reset (visit' xs)
```

Theoretical pearl: A simple proof of a folklore theorem about delimited control, Biernacki et al. JFP 16

Traverse/reverse with control and shift

```
let visit xs =
  match xs with
  | [] -> []
  | x :: xs1 ->
    visit (shift k (x :: k xs))

let traverse xs = reset (visit xs)
```

```
let visit' xs =
  match xs with
  | [] -> []
  | x :: xs1 ->
    visit' (control k (x :: k xs))

let reverse xs = reset (visit' xs)
```

$$copy(xs) = \\ \quad \mathbf{ens}\ xs{=}[]; \mathbf{ret}\ [] \\ \vee\ \exists\, x\ xs_1.\ \mathbf{ens}\ xs{=}x :: xs_1; copy(xs_1); r_1.\ \mathbf{ret}\ x :: r_1$$

$$rev(xs, acc) = \\ \quad \mathbf{ens}\ xs{=}[]; \mathbf{ret}\ acc \\ \vee\ \exists\, x\ xs_1.\ \mathbf{ens}\ xs{=}x :: xs_1; rev(xs_1, x :: acc)$$

$$traverse(xs) \sqsubseteq copy(xs)$$

$$reverse(xs) \sqsubseteq rev(xs, [])$$

Traverse/reverse with control and shift

$$traverse(xs) \sqsubseteq copy(xs)$$

Easy, due to the delimiter that **shift** leaves around the recursive call.

$$reverse(xs) \sqsubseteq rev(xs, [])$$

Quite a bit tricker: requires generalisation over the *continuation* of the recursive call, due to the lack of delimiter in **control**.

Evaluation

Table 1. Experimental Results. (LoC and LoS stand for lines of code and lines of specification, respectively)

Program	**Desc.**	**LoC**	**LoS**	**Total(s)**	**SMT(s)**
State monad [Asai and Kiselyov 2011]	HO	21	2	4.84	0.02
Alice-Cat [Materzok and Biernacki 2011]	Multi	3	1	0.33	0.02
Printf [Asai and Kiselyov 2011]	ATM	30	6	2.20	0.05
Flip [Danvy and Filinski 1990]	Multi	9	1	3.89	0.04
Toss [Song et al. 2024]	Imp	8	1	6.50	0.08
HTTcc-inc3 [Delbianco and Nanevski 2013]	Imp	5	1	5.05	0.05
Times [Asai and Kiselyov 2011]	Ind	17	5	0.88	0.11
Total		93	17	23.69	0.37

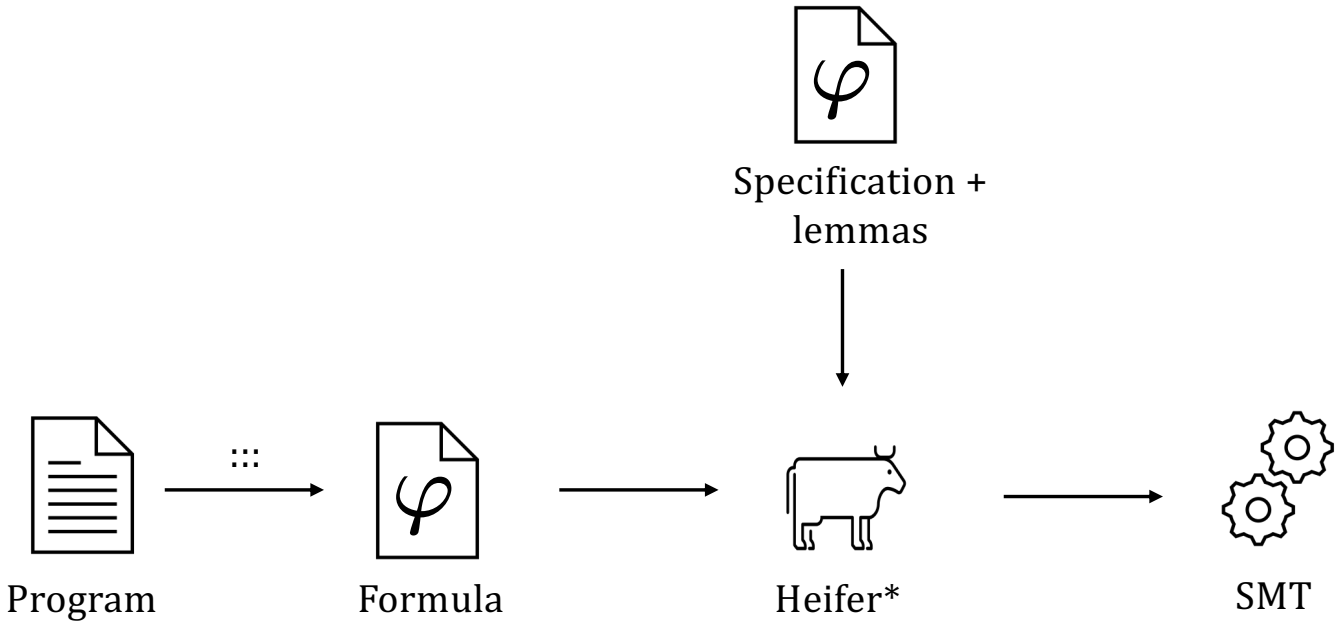
Conclusion

Deferring abstraction allowed us to avoid non-context-local reasoning.

Outline

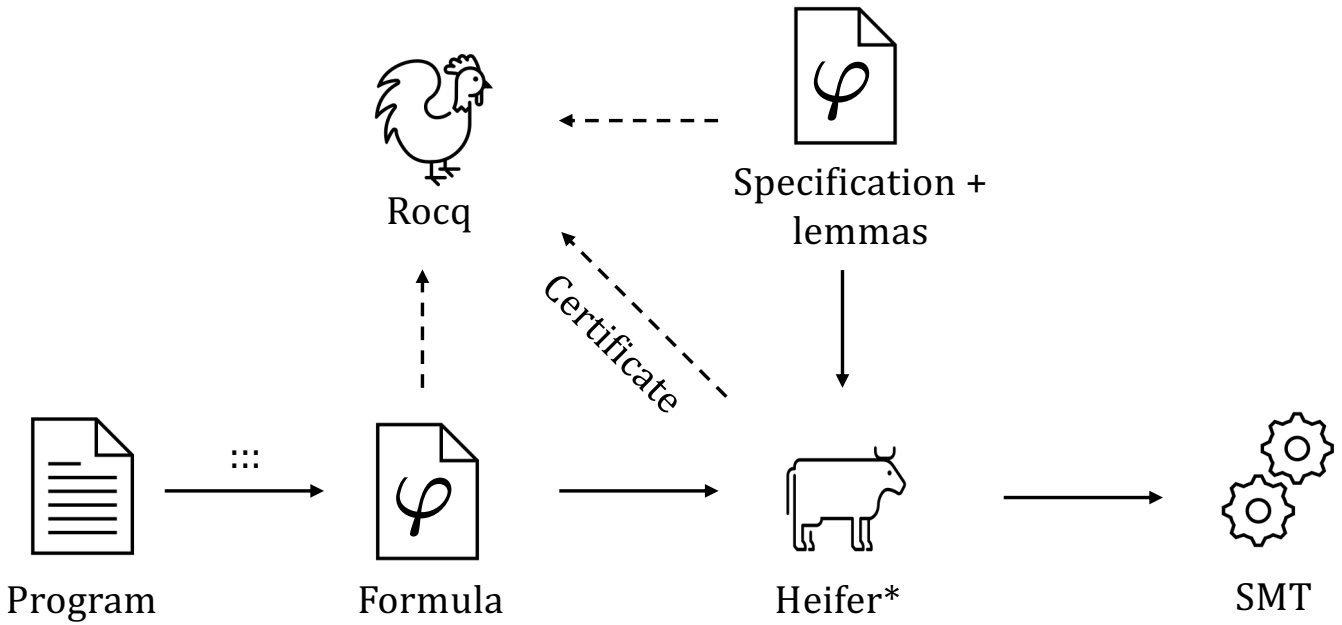
- Reasoning about imperative higher-order programs
- Reasoning about effect handlers
- Reasoning about delimited continuations
- Mechanising staged logic
- Conclusion

An automated verifier for staged logic



*Higher-order Effectful Imperative Function Entailments & Reasoning

What we would like



*Higher-order Effectful Imperative Function Entailments & Reasoning

Staged logic proofs

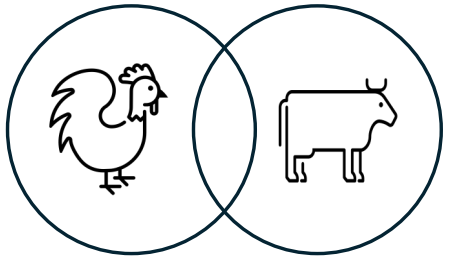
Rocq sequent

Staged logic 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))]))
```

State of the mechanisation

- The mechanisation handles many verification problems that Heifer does, as well as some it does not
- Some problems are still handled by Heifer only, due to limitations of the encoding
  - “Deep” entailments (between continuations or functions in the store) are a work in progress, using step-indexing
- Extensive use of generalised rewriting
- Tactics for symbolic execution



Outline

- Reasoning about imperative higher-order programs
- Reasoning about effect handlers
- Reasoning about delimited continuations
- Mechanising staged logic
- Conclusion

Conclusion

Staged logic: a refinement-based program logic for automated verification of effectful higher-order programs

Key insight: deferring abstraction by keeping specifications precise, allowing one to avoid auxiliary specifications such as:

- Invariants
- Protocols, handler invariants
- Non-context-local triples

Conclusion

https://github.com/hipsleek/Heifer/
https://github.com/dariusf/staged

Staged logic: a refinement-based program logic for automated verification of effectful higher-order programs

- Remaining work
  - Delimited continuations paper (work in progress)
- Future work
  - A step-indexed model
  - Certification

Thanks for listening!

```
# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# List.fold_right (fun c x -> c :: x) [1; 10; 100] [1000];;
- : int list = [1; 10; 100; 1000]

# let x = ref [1000] in (List.fold_right (fun c () -> x := c
    :: !x) [1; 10; 100] (); !x);;
- : int list = [1; 10; 100; 1000]

# List.fold_right (fun c r -> c + r) [1; 10; 100] 0;;
- : int = 111
```

```
# let r = ref 0 in (List.fold_right (fun c () -> r := c +
    !r) [1; 10; 100] (); !r);;
- : int = 111

# List.fold_right (fun c (x, r) -> (c :: x, c + r)) [1; 10;
    100] ([1000], 0);;
- : int list * int = ([1; 10; 100; 1000], 111)

# let x = ref [1000] and r = ref 0 in (List.fold_right (fun
    c () -> (x := c :: !x; r := c + !r)) [1; 10; 100] ();
    (!x, !r));;
- : int list * int = ([1; 10; 100; 1000], 111)
```

Publications

Topic	Papers
Higher-order functions	FM 24
Effect handlers	APLAS 22, ICFP 24
Delimited continuations + mechanisation	Under submission

Traverse using shift

The statement using shift can be proved directly by induction.

$$traverse(xs) \sqsubseteq copy(xs)$$

Inductive case $\langle visit(x :: xs_1)\rangle \sqsubseteq copy(x :: xs_1)$

The delimiter in the makes it easy to apply the induction hypothesis

$$
\begin{array}{lll}
 & \langle visit(x :: xs_1)\rangle & \\
\sqsubseteq & \langle (\textbf{shift}\ k.\ k(xs_1); r.\ \textbf{ret}\ x :: r); r_1.\ visit(r_1)\rangle & \text{Unfold} \\
\sqsubseteq & \textbf{ens}\ k{=}(\lambda r_1.\ \langle visit(r_1)\rangle); \langle k(xs_1); r.\ \textbf{ret}\ x :: r\rangle & \text{Reduction}^2 \\
\sqsubseteq & \langle \langle visit(xs_1)\rangle; r.\ \textbf{ret}\ x :: r\rangle & \text{Simplify} \\
\sqsubseteq & \langle copy(xs_1); r.\ \textbf{ret}\ x :: r\rangle & \text{Ind. hyp.} \\
\sqsubseteq & copy(xs_1); r.\ \textbf{ret}\ x :: r & \text{Pure}
\end{array}
$$

Reverse using control

The statement using control is tricker.

$$reverse(xs) \sqsubseteq rev(xs, acc)$$

In the inductive case, we eventually get stuck in this state:

$$\langle visit'\,(xs)\rangle \sqsubseteq rev(xs, [\,])$$
$$\vdots$$
$$\langle visit'\,(xs_1); r.\ \mathbf{ret}\ x :: r\rangle \sqsubseteq \ldots$$

We can't appeal to the induction hypothesis because of the lack of delimiter.

Reverse using control

$$\langle visit'\,(xs)\rangle \sqsubseteq rev(xs, [])$$
$$\vdots$$
$$\langle visit'\,(xs_1); r.\ \mathbf{ret}\ x :: r\rangle \sqsubseteq \ldots$$

We need the following strengthening, where we generalise over the continuation of visit'.

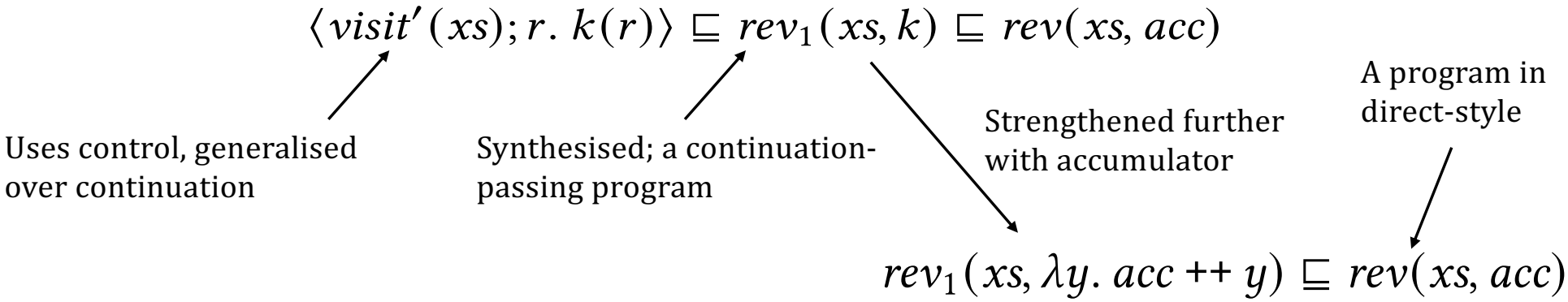
$$\langle visit'\,(xs); r.\ k(r)\rangle \sqsubseteq \ldots$$

This allows us to talk about what visit' captures.

Proof sketch

To prove $\langle visit'(xs) \rangle \sqsubseteq rev(xs, [])$,

We prove the chain of statements:



Why does the proof work?

Inductive case $\langle visit(x :: xs_1)\rangle \sqsubseteq copy(x :: xs_1)$

$\langle visit(x :: xs_1)\rangle$

$\sqsubseteq$ $\langle (\textbf{shift}\ k.\ k(xs_1); r.\ \textbf{ret}\ x :: r); r_1.\ visit(r_1)\rangle$ Unfold

$\sqsubseteq$ $\textbf{ens}\ k{=}(\lambda r_1.\ \langle visit(r_1)\rangle); \langle k(xs_1); r.\ \textbf{ret}\ x :: r\rangle$ Reduction[2]

$\sqsubseteq$ $\langle \langle visit(xs_1)\rangle; r.\ \textbf{ret}\ x :: r\rangle$ Simplify

$\sqsubseteq$ $\langle copy(xs_1); r.\ \textbf{ret}\ x :: r\rangle$ Ind. hyp.

$\sqsubseteq$ $copy(xs_1); r.\ \textbf{ret}\ x :: r$ Pure

visit cannot capture it, due to the delimiter

part of the meta-context

visit' is not delimited and can capture the rest of the computation

this is part of the inductive argument; visit' *repeatedly* captures it and nests it one level deeper, resulting in the reversal

Inductive case $\langle visit'(x :: xs_1); r.\ k(r)\rangle \sqsubseteq rev_1(x :: xs_1, k)$

$\langle visit'(x :: xs_1); r.\ k(r)\rangle$

$\sqsubseteq$ $\langle (\textbf{control}\ k_1.\ k_1(xs_1); r.\ \textbf{ret}\ x :: r); r_1.\ visit'(r_1); r.\ k(r)\rangle$ Unfold

$\sqsubseteq$ $\textbf{ens}\ k_1{=}(\lambda r_1.\ visit'(r_1); ; r.\ k(r)); \langle k_1(xs_1); r.\ \textbf{ret}\ x :: r\rangle$ Reduction[3]

$\sqsubseteq$ $\langle visit'(xs_1); r_1.\ k(r_1); r.\ \textbf{ret}\ x :: r\rangle$ Simplify

$\sqsubseteq$ $rev_1(xs_1, \lambda r_1.\ k(r_1); r.\ \textbf{ret}\ x :: r)$ Ind. hyp.

stuck

Semantics of staged logic

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

heaps result

$\mathcal{E}, h_1, h_2, v \vDash \mathbf{req}\, P\, \varphi$ *if*

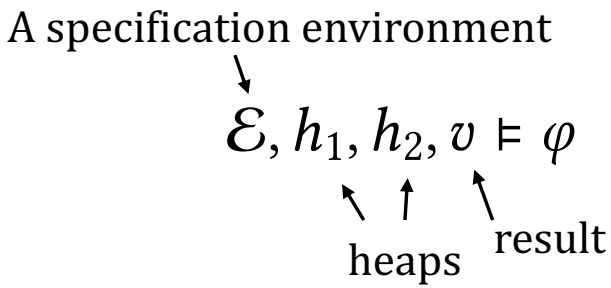
$$\forall h_p\ h_r, (h_p \vDash P \text{ and } h_1 = h_p \circ h_r) \Longrightarrow \mathcal{E}, h_r, h_2, v \vDash \varphi$$

$\mathcal{E}, h_1, h_2, v \vDash \mathbf{ens}\, Q$ *if*

$$\exists h_3, (h_3, v \vDash Q) \text{ and } h_2 = h_1 \circ h_3$$

Internalisation of the operational behaviour of heap entailment

Semantics of staged logic



$$\mathcal{E}, h_1, h_2, v \vDash f(a) \ \textit{if}$$
$$\mathcal{E}, h_1, h_2, v \vDash \mathcal{E}[f](a)$$

Encoding staged logic

A specification environment

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

heaps result

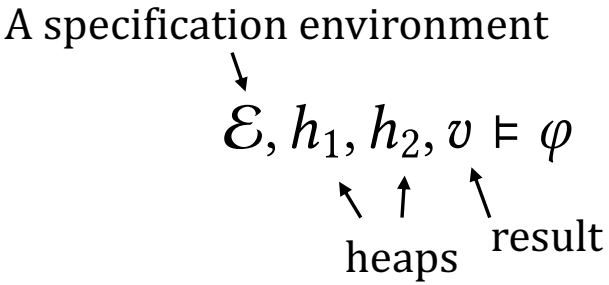
Separation logic: `heap -> Prop`

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



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.

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

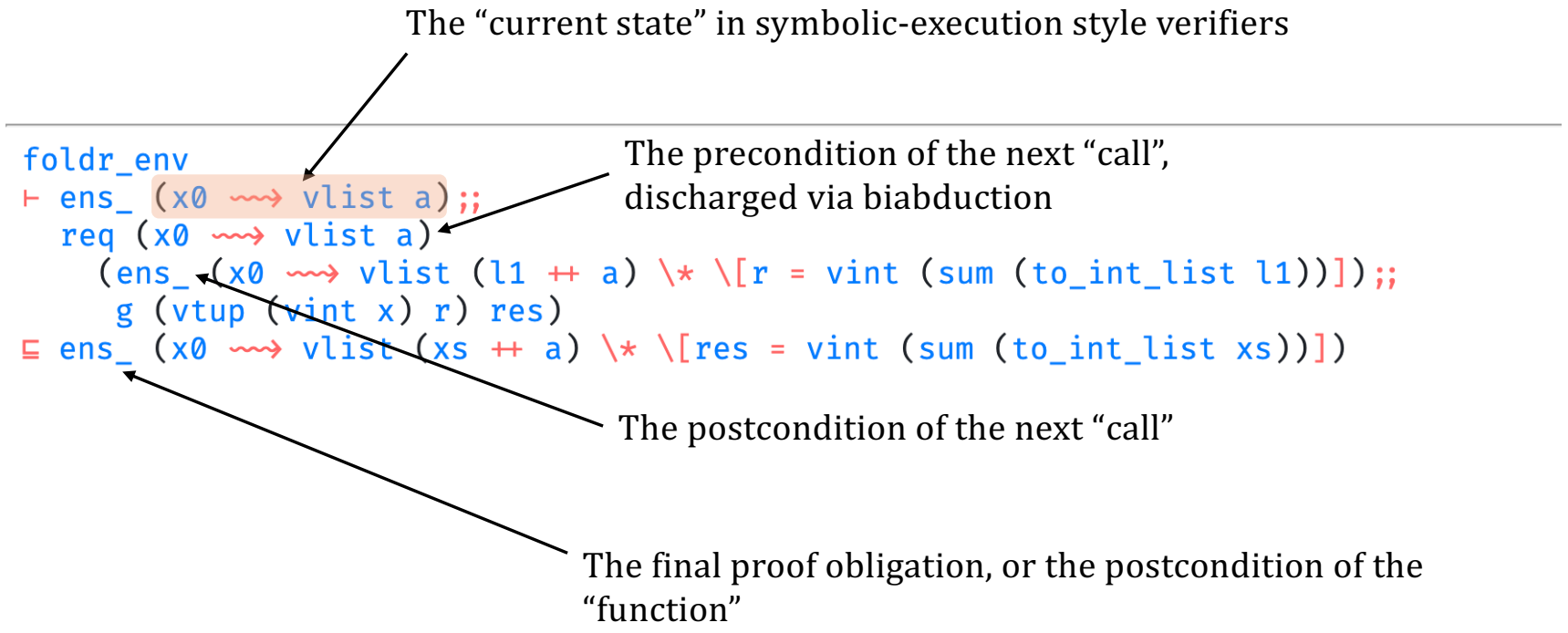
```
Lemma entails_ens : forall H1 H2,
  H1 ==> H2 ->
  entails (ens H1) (ens H2).
```

Staged logic proofs

```
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))]))
```

Contravariant; can be moved to the left side

Staged logic proofs



Staged logic proofs

```
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))])
```

Staged logic proofs

```
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)))]
```

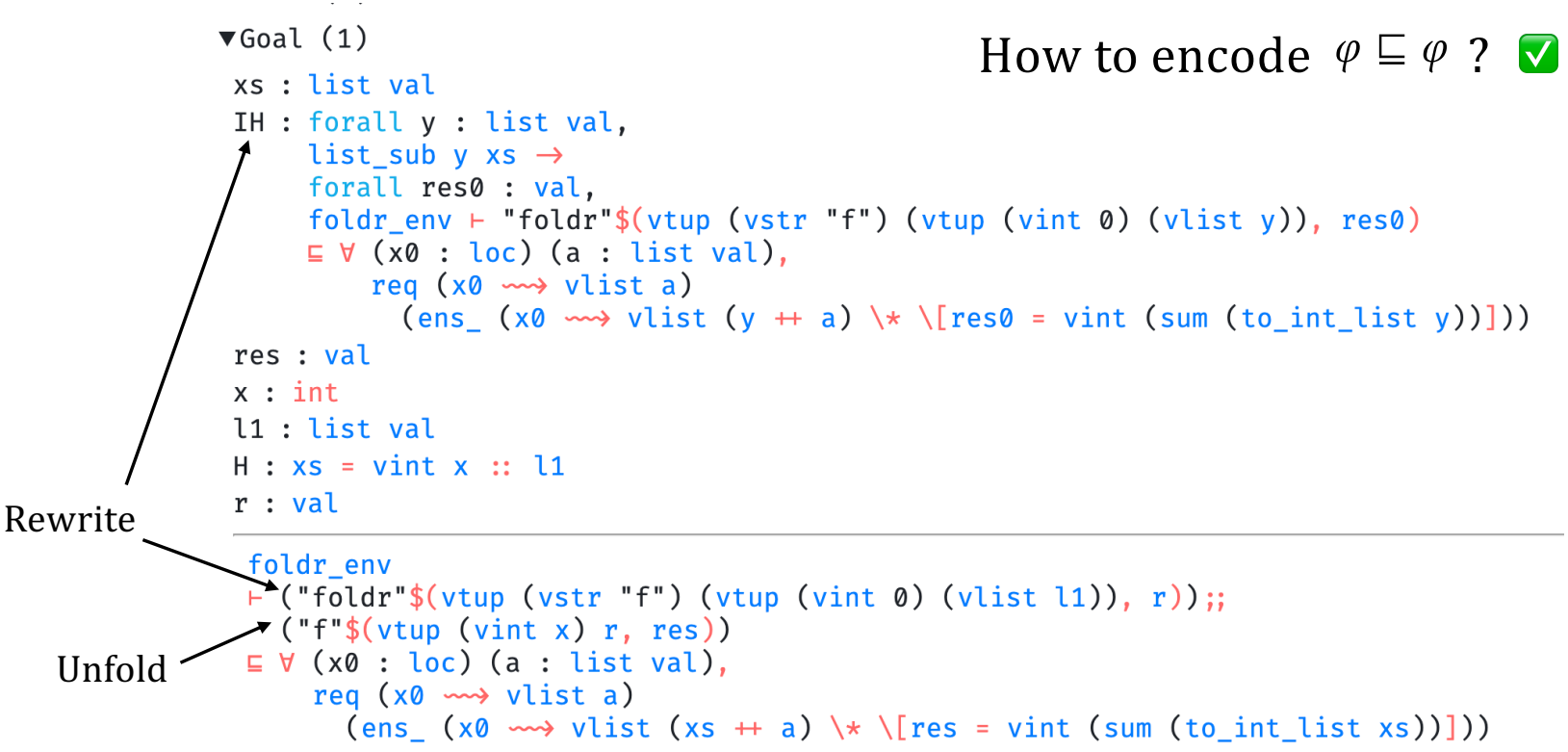
Staged logic proofs

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

Staged logic proofs

```
No more goals
```

What we would like

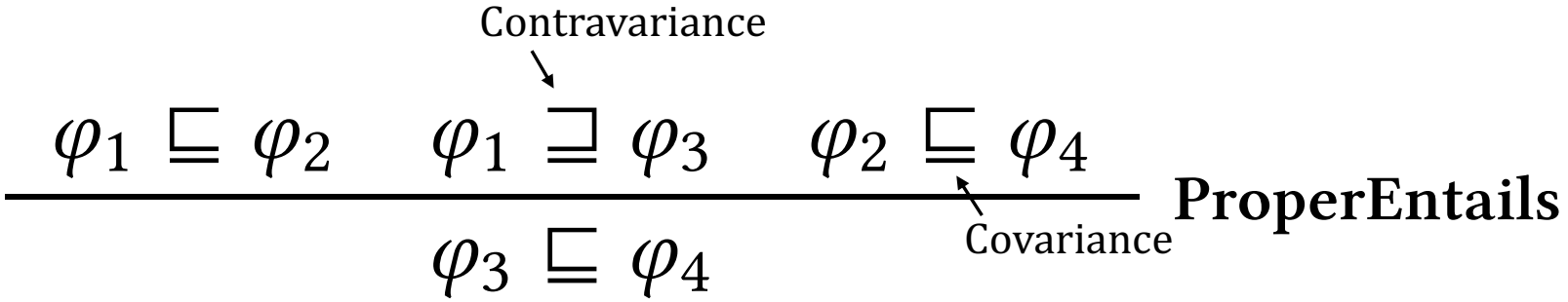


How to encode $\varphi \sqsubseteq \varphi$? ✅

Rewriting

- We use Rocq's *generalised rewriting*, with entails as the relation

- entails must be shown to be *proper* in both arguments, respecting the relations that we want to rewrite under



What we would like

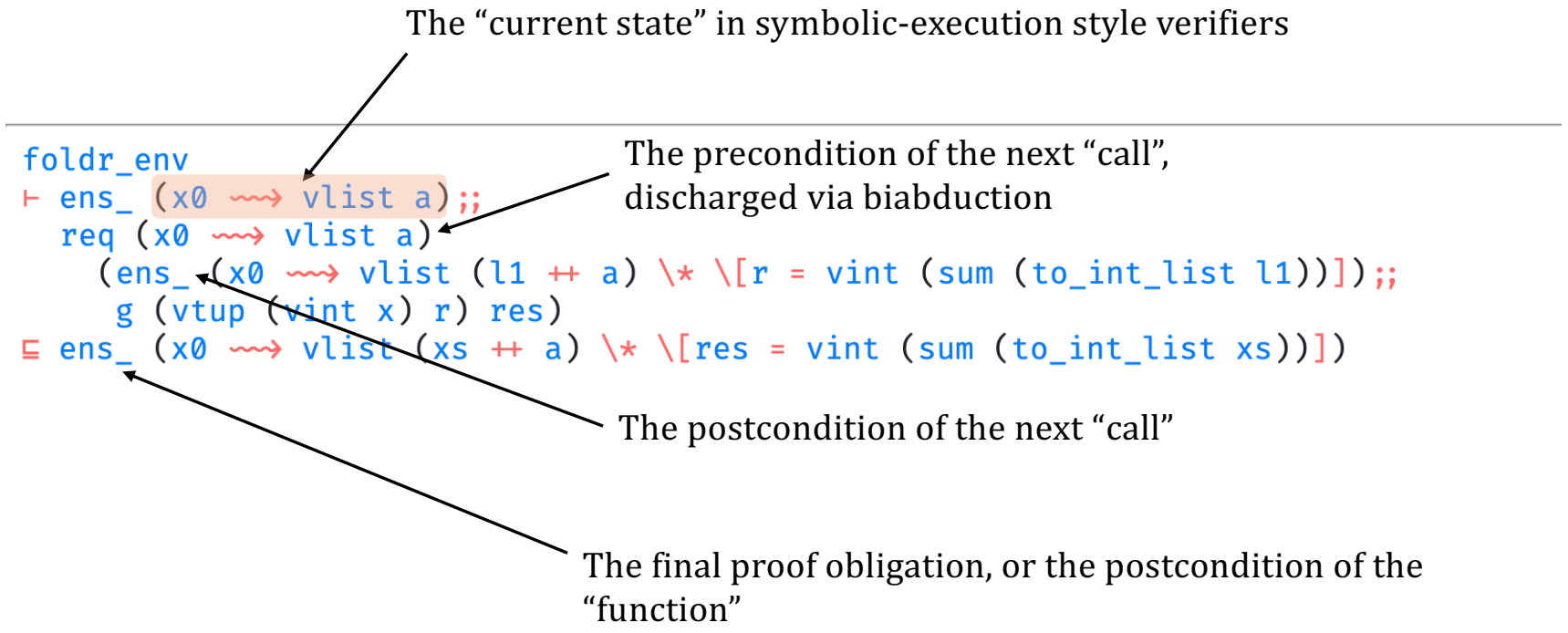
Rewriting ✅

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

```
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))]))
```

Contravariant; can be moved to the left side

What we would like



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

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
- Certify: validate implementation, not just ideas
- Support new work: work manually on harder proofs, broaden fragment that can be fully automated