Staged Specification Logic

Higher-order Imperative Programs + Algebraic Effects

Yahui Song

Research Fellow @ National University of Singapore (NUS)

September 2024

My Research

- PhD (2018 Aug – 2023 May)

Thesis: Symbolic Temporal Verification Techniques with Extended Regular Expressions

Keywords: Modularly (Scalability), Expressive Specification, Hoare-style Verification

Applications
- Event-based reactive systems [ICFEM 2020]
- Synchronous languages like Esterel [VMCAI 2021]
- User-defined algebraic effects and handlers [APLAS 2022]
- Real-time systems [TACAS 2023]

- Research Fellow (2023 – now)

My Research

- PhD (2018 Aug – 2023 May)

Thesis: Symbolic Temporal Verification Techniques with Extended Regular Expressions

Keywords: Modularly (Scalability), Expressive Specification, Hoare-style Verification

Applications
- Event-based reactive systems [ICFEM 2020]
- Synchronous languages like Esterel [VMCAI 2021]
- User-defined algebraic effects and handlers [APLAS 2022]
- Real-time systems [TACAS 2023]

- Research Fellow (2023 – now)

  - Temporal Property Guided Bug Detection and Repair [FSE 2024]

  - **Staged Specification Logic:**

    Higher-order Imperative Programs [FM 2024]

    Unrestricted Algebraic Effects and Handling [ICFP 2024]

Staged Specification Logic for Verifying Higher-Order Imperative Programs

Darius Foo, Yahui Song, Wei-Ngan Chin

NUS National University of Singapore

FM Milano 2024

Challenges with Effectful Higher-order Functions

- Existing (automated) verifier varies greatly:
  - Pure only: Dafny, WhyML, Cameleer
  - Type system (Rust) guarantees: Creusot, Prusti
  - Interactive: Iris, CFML, Pulse/Steel (F*)
- When supported, specifications are *imprecise*
- Is there a ***precise* and *general*** way to support effectful higher-order functions in ***automated* program verifiers**?

Specification in Iris

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

f must preserve the invariant

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

(Separation logic) Invariant relating input to result

foldr should not change the list

$$foldr\ f\ a\ l$$

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

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

The Use of Abstract Properties

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

- `foldr` commits to an *abstraction* of f's behavior
- The 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 (page 32)

Difficult to handle cases like:

- Problem 1: Mutable list

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

- Problem 2: Strengthened precondition

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

- Problem 3: Exceptions/effects

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

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

$$foldr\ f\ a\ l$$

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

Difficult to handle cases like:

- Problem 1: Mutable list

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

- Problem 2: Strengthened precondition

```
let
```

- P

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

Can we get rid of abstraction when designing spec for HO-functions?

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

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

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

We Propose "Staged Specification Logic"

Sequencing (Un)interpreted relations

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

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

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

$\Longrightarrow$ *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
```

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

We can no longer assume anything about y at this point.

We also cannot assume anything about x!

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)
```

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

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

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

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

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

$$\pi \ ::= \ \ldots \mid \varphi \sqsubseteq \varphi$$

3. Re-summarization of recursion via lemmas

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

```
let foldr_sum_state x xs init


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

3. Re-summarization of recursion via lemmas

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

```
let foldr_sum_state x xs init
```

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

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

$$sum(li, res) = \\ \quad l{=}[] \wedge res{=}0 \\ \vee\ \exists r, l_1 \cdot l{=}x{::}l_1 \wedge sum(l_1, r) \wedge res{=}x{+}r$$

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

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

3. Re-summarization of recursion via lemmas

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

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

- Recovering abstraction: proving entailments

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

Inductive Step

3. Re-summarization of recursion via lemmas

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

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

- Recovering abstraction: proving entailments

Bi-abduction

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

$$\dfrac{}{x \mapsto i \vdash \exists\, i \,.\, x \mapsto i * emp}\ \text{SL} \qquad \dfrac{}{\begin{array}{c} x \mapsto i{+}r{+}h \wedge res{=}h{+}r{+}init \wedge r{=}sum(\ldots \\ \vdash \exists\, r \,.\, x \mapsto i{+}r \wedge res{=}r{+}init \wedge r{=}sum(h{::}t) \end{array}}$$

ENTAIL

$$\exists\, i \,.\, \mathbf{req}\, x \mapsto i ; \exists\, r, h, t \,.\, \mathbf{ens}\, x \mapsto i{+}r{+}h \wedge res{=}h{+}r{+}init \wedge r{=}sum(t) \wedge xs{=}h{::}t$$

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

NORMALIZE

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

UNFOLD

$$\exists\, r_1, h, t \,.\, \mathbf{ens}\, xs{=}h{::}t ; \exists\, i, r \,.\, \mathbf{req}\, x \mapsto i ; \mathbf{ens}\, x \mapsto i{+}r \wedge r_1{=}r{+}init \wedge r{=}sum(t) ; g(h, r_1, res) \sqsubseteq \ldots$$

REWRITE

$$(\ldots \vee \exists\, r_1, h, t \,.\, \mathbf{ens}\, xs{=}h{::}t ; foldr(g, t, init, r_1) ; g(h, r_1, res)) \sqsubseteq \ldots$$

UNFOLD

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

3. Re-summarization of recursion via lemmas

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

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

- Recovering abstraction: proving entailments

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

Solutions with Staged Logic

- Problem 1: Mutable list

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

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

- Problem 2: Strengthened precondition

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

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

- Problem 3: Exceptions/effects

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

More about Effects later!

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

Implementation & Evaluation

- 5K LoC on top of OCaml 5
- Reasonably low verification time
- Feasibility & increased expressiveness over existing systems

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	

inexpressible

incomparable

LoS/LoC = 0.37 **= 2.49** **= 0.73**

Summary

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

- Staged logic for effectful higher-order programs

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

    $\Longrightarrow$ *Defer abstraction* until appropriate

- Heifer – a new automated verifier: https://github.com/hipsleek/heifer

*Higher-order Effectful Imperative Function Entailments and Reasoning

Specification and Verification for Unrestricted Algebraic Effects and Handling

Yahui Song, Darius Foo, Wei-Ngan Chin

User-defined Effects and Handlers

This prints: 0 1 2 3 4

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string “4 "
```

effect declaration

suspends current computation

computation

delimited continuation

handler

resume suspended computation

Motivation Example

```
1 effect Label: int
2 (* User-defined effect, which will be resumed with int values *)
3
4 let callee () : int
5 = let x = ref 0 in                 (* initialize x to zero *)
6   let ret = perform Label in       (* the handler has no access to x *)
7   x := !x + 1;                     (* increment x from zero to one *)
8   assert (!x = 1);                 (* x now contains one *)
9   ret + 2                          (* return the resumed value + 2 *)
```

- Zero-shot handlers: abandon the continuation, just like exception handlers;
- One-shot handlers: resume the continuation once, the assertion on line 8 succeeds;
- Multi-shot handlers: resume the continuation more than once, the assertion on line 8 fails.

Motivation Example

```
effect Label: int
(* User-defined effect, which will be resumed with int values *)

let callee () : int
= let x = ref 0 in                  (* initialize x to zero *)
  let ret = perform Label in        (* the handler has no access to x *)
  x := !x + 1;                      (* increment x from zero to one *)
  assert (!x = 1);                  (* x now contains one *)
  ret + 2                           (* return the resumed value + 2 *)
```

Existing verification techniques:

- **multi-shot continuations + pure setting, e.g. [Song et al. 2022];**
- **heap manipulation + one-shot continuations, e.g. [de Vilhena and Pottier 2021];**
- **multi-shot + heap-manipulation, under a restricted frame rule, e.g. [de Vilhena 2022].**

Protocol Based Approach [de Vilhena and Pottier 2021]

- Hazel & Maze: Model client-handler interactions in the form of protocols
- Globally define the effects that clients perform and the replies they receive from handlers
- Global assumptions to provide explicit (or early) interpretation effects

$$\mathit{ABORT} \triangleq !\ (()) \ \{\mathsf{True}\}.\ ?\, y\ (y)\ \{\mathsf{False}\}$$

$$\mathit{CXCHG} \triangleq !\, x\, x'\ (x')\ \{\ell \mapsto x\}.\ ?\ (x)\ \{\ell \mapsto x'\}$$

$$\mathit{AXCHG} \triangleq !\, x\, x'\ (x')\ \{I\ x\}.\ ?\ (x)\ \{I\ x'\}$$

$$\begin{aligned}\mathit{READWRITE} \triangleq \quad & \mathit{read}\ \#\ !\, x\ (())\ \{I\ x\}.\ ?\ (x)\ \{I\ x\} \\ & + \mathit{write}\ \#\ !\, x\, x'\ (x')\ \{I\ x\}.\ ?\ (())\ \{I\ x'\}\end{aligned}$$

Motivation Example

```
effect Label: int
(* User-defined effect, which will be resumed with int values *)

let callee () : int
= let x = ref 0 in                    (* initialize x to zero *)
  let ret = perform Label in          (* the handler has no access to x *)
  x := !x + 1;                        (* increment x from zero to one *)
  assert (!x = 1);                    (* x now contains one *)
  ret + 2                             (* return the resumed value + 2 *)
```

Existing verification techniques:

- **multi-shot continuations + pure setting, e.g. [Song et al. 2022];**
- **heap manipulation + one-shot continuations, e.g. [de Vilhena and Pottier 2021];**
- **multi-shot + heap-manipulation, under a restricted frame rule, e.g. [de Vilhena 2022].**

Specification and Verification for

Unrestricted Algebraic Effects and Handling

Yahui Song, Darius Foo, Wei-Ngan Chin

4th Sep @ ICFP 2024, Milan, Italy

Our Solution: Effectful Specification Logic (ESL)

- Fully modular per-method verification (no global assumption)
- Sequencing, $\varphi_1 ; \varphi_2$
- Uninterpreted relations for ***unhandled effects*** and unknown functions, $E(x, r)$

result

input

- Reducible ***try-catch logic constructs***
- *Normalization: compact* each sequence of pre/post stages, via bi-abduction
- Use ***re-summarization*** (lemma) when handling recursive generated effects

$$\boxed{\begin{array}{llll} (ESL) & \varphi & ::= & \mathbf{req}\ P \mid \mathbf{ens}[r]\ Q \mid \varphi;\varphi \mid \varphi \vee \varphi \mid \exists\, x^*;\varphi \mid \\ & & & E(x,r) \mid f(x^*, r) \mid \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi \end{array}}$$

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

We propose ESL

$$(ESL) \qquad \varphi \quad ::= \quad \mathbf{req}\ P \mid \mathbf{ens}[r]\ Q \mid \varphi;\varphi \mid \varphi \vee \varphi \mid \exists\, x^*; \varphi \mid$$
$$\color{red}{E}(x, r) \mid f(x^*, r) \mid \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$$

```
1  effect Label: int
2  (* User-defined effect, which will be resumed with int values *)
3
4  let callee () : int
5  = let x = ref 0 in              (* initialize x to zero *)
6    let ret = perform Label in    (* the handler has no access to x *)
7    x := !x + 1;                  (* increment x from zero to one *)
8    assert (!x = 1);              (* x now contains one *)
9    ret + 2                       (* return the resumed value + 2 *)
```

$$\begin{aligned} callee(r_c) \;=\; & \exists x \cdot \mathbf{ens}\ \ x{\mapsto}0\ ; && //\ \text{Line 5} \\ & \exists ret \cdot Label(ret)\ ; && //\ \text{Line 6} \\ & \exists z \cdot \mathbf{req}\ \ x{\mapsto}z \wedge z{+}1{=}1\ \ \mathbf{ens}[r_c]\ x{\mapsto}z{+}1 \wedge r_c{=}(ret{+}2) && //\ \text{Lines 7-9} \end{aligned}$$

We propose ESL

$$(ESL) \qquad \varphi \quad ::= \quad \mathbf{req}\ P \mid \mathbf{ens}[r]\ Q \mid \varphi;\varphi \mid \varphi \vee \varphi \mid \exists\, x^*;\varphi \mid E(x,r) \mid f(x^*, r) \mid \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$$

```
1  effect Label: int
2  (* User-defined effect, which will be resumed with int values *)
3
4  let callee () : int
5  = let x = ref 0 in                  (* initi
6    let ret = perform Label in        (* the h
7    x := !x + 1;                      (* incre
8    assert (!x = 1);                  (* x now
9    ret + 2                           (* retur
```

Bi-abduction:

∃ z; req x→z;
      ens x→z+1;
∃ b; req x→b ∧ b=1

$$\begin{aligned} callee(r_c) \;=\; & \exists x \cdot \mathbf{ens}\ \ x{\mapsto}0\ ; && \text{// Line 5} \\ & \exists ret \cdot Label(ret)\ ; && \text{// Line 6} \\ & \exists z \cdot \mathbf{req}\ \ x{\mapsto}z \wedge z{+}1{=}1\ \ \mathbf{ens}[r_c]\ x{\mapsto}z{+}1 \wedge r_c{=}(ret{+}2) && \text{// Lines 7-9} \end{aligned}$$

We propose ESL

$$
\begin{array}{llll}
(ESL) & \varphi & ::= & \mathbf{req}\ P \mid \mathbf{ens}[r]\ Q \mid \varphi;\varphi \mid \varphi \vee \varphi \mid \exists\, x^*; \varphi \mid \\
 & & & E(x, r) \mid f(x^*, r) \mid \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi
\end{array}
$$

```
1  effect Label: int
2  (* User-defined effect, which will be resumed with int values *)
3
4  let callee () : int
5  = let x = ref 0 in             (* initi
6    let ret = perform Label in   (* the h
7    x := !x + 1;                 (* incre
8    assert (!x = 1);             (* x now
9    ret + 2                      (* return the resumed value + 2 *)
```

Bi-abduction:

$\exists\ z;\ \mathrm{req}\ x \to z \wedge z{+}1{=}1;$
$\quad\quad \mathrm{ens}\ x \to z{+}1$

$$
\begin{array}{lll}
callee(r_c) = & \exists x \cdot \mathbf{ens}\ \ x {\mapsto} 0\,; & //\ \text{Line 5} \\
 & \exists ret \cdot Label(ret)\,; & //\ \text{Line 6} \\
 & \exists z \cdot \mathbf{req}\ \ x {\mapsto} z \wedge z{+}1{=}1\ \ \mathbf{ens}[r_c]\ x {\mapsto} z{+}1 \wedge r_c{=}(ret{+}2) & //\ \text{Lines 7-9}
\end{array}
$$

Try-Catch Reduction (Examples)

$$\begin{aligned}
\mathit{callee}(r_c) \;=\; & \exists x \cdot \mathbf{ens}\;\; x \mapsto 0\;; && \text{// Line 5}\\
& \exists \mathit{ret} \cdot \mathit{Label}(\mathit{ret})\;; && \text{// Line 6}\\
& \exists z \cdot \mathbf{req}\;\; x \mapsto z \wedge z{+}1{=}1\;\; \mathbf{ens}[r_c]\; x \mapsto z{+}1 \wedge r_c{=}(\mathit{ret}{+}2) && \text{// Lines 7-9}
\end{aligned}$$

```
let zero_shot () : int
(* zero_shot(r_z) = ∃x ; ens[r_z] x↦0 ∧ r_z=-1 *)
= match callee () with
| effect Label k -> -1
```

```
let one_shot () : int
(* one_shot(r_o) = ∃x ; ens[r_o] x↦1 ∧ r_o=5 *)
= match callee () with
| effect Label k -> resume k 3
```

Try-Catch Reduction (Examples)

$$
\begin{aligned}
\mathit{callee}(r_c) \;=\;& \exists x \cdot \mathbf{ens}\;\; x{\mapsto}0\;; && \text{// Line 5}\\
& \exists ret \cdot \mathit{Label}(ret)\;; && \text{// Line 6}\\
& \exists z \cdot \mathbf{req}\;\; x{\mapsto}z \wedge z{+}1{=}1\;\; \mathbf{ens}[r_c]\; x{\mapsto}z{+}1 \wedge r_c{=}(ret{+}2) && \text{// Lines 7-9}
\end{aligned}
$$

```
let zero_shot () : int
(* zero_shot(r_z) = ∃x ; ens[r_z] x↦0 ∧ r_z=-1 *)
= match callee () with
| effect Label k -> -1
```

```
let one_shot () : int
(* one_shot(r_o) = ∃x ; ens[r_o] x↦1 ∧ r_o=5 *)
= match callee () with
| effect Label k -> resume k 3
```

```
let multi_shot () : int
(* multi_shot(r_m) = req false *)
= match callee () with
| effect Label k ->
  let _ = resume k 4 in resume k 5
```

Try-Catch Reduction (Examples)

$$
\begin{aligned}
callee(r_c) \;=\; & \exists x \cdot \mathbf{ens}\;\; x{\mapsto}0\;; && //\ \text{Line 5}\\
& \exists ret \cdot Label(ret)\;; && //\ \text{Line 6}\\
& \exists z \cdot \mathbf{req}\;\; x{\mapsto}z \wedge z{+}1{=}1\;\; \mathbf{ens}[r_c]\; x{\mapsto}z{+}1 \wedge r_c{=}(ret{+}2) && //\ \text{Lines 7-9}
\end{aligned}
$$

```
let zero_shot () : int
(* zero_shot(r_z) = ∃x ; ens[r_z] x↦0 ∧ r_z=-1 *)
= match callee () with
| effect Label k -> -1
```

```
let one_shot () : int
(* one_shot(r_o) = ∃x ; ens[r_o] x↦1 ∧ r_o=5 *)
= match callee () with
| effect Label k -> resume k 3
```

```
let multi_shot () : int
(* multi_shot(r_m) = req false *)
= match callee () with
| effect Label k ->
  let _ = resume k 4 in resume k 5
```

Intuition:

- Explicit access to continuation
- Modular verification:
  - try-catch reduction
  - normalization via bi-abduction

Try-Catch Reduction (Selected Rules)

- The base case:

$$\frac{(x \rightarrow \Phi_n) \in \mathcal{H}_\Phi}{\boxed{\mathbf{try}[\delta](\mathcal{N}[r])\ \mathbf{catch}\ \mathcal{H}_\Phi} \rightsquigarrow \boxed{\mathcal{N}[r]\ ;\ \Phi_n[r/x]}}\ [\mathcal{R}\text{-}Normal]$$

- When handling an effect, first reason about the behaviours of its continuation

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x,r) \qquad E \in dom(\mathcal{H}_\Phi) \qquad \mathbf{try}[d](\theta)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \Phi}{\mathbf{try}[d](\mathcal{E}\ ;\ \theta)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathbf{try}[d](\mathcal{E} \# \Phi)\ \mathbf{catch}\ \mathcal{H}_\Phi}\ [\mathcal{R}\text{-}Deep]$$

- Instantiate the high-order predicate k using the continuation's specification

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x,r) \qquad (E(y)k \rightarrow \Phi) \in \mathcal{H}_\Phi \qquad \Phi' = \Phi[x/y, (\lambda(r, r_c) \rightarrow \Phi[r_c])/k]}{\mathbf{try}[\delta](\mathcal{E} \# \Phi[r_c])\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{N}\ ;\ \Phi'}\ [\mathcal{R}\text{-}Eff\text{-}Handle]$$

Try-Catch Reduction (Selected Rules)

- The base case:

$$\frac{(x \to \Phi_n) \in \mathcal{H}_\Phi}{\mathbf{try}[\delta](\mathcal{N}[r])\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{N}[r]\ ;\ \Phi_n[r/x]}\ [\mathcal{R}\text{-}Normal]$$

- When handling an effect, first reason about the behaviours of its continuation

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x, r) \qquad E \in dom(\mathcal{H}_\Phi) \qquad \mathbf{try}[d](\theta)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \Phi}{\mathbf{try}[d](\mathcal{E}\ ;\ \theta)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathbf{try}[d](\mathcal{E}\ \#\ \Phi)\ \mathbf{catch}\ \mathcal{H}_\Phi}\ [\mathcal{R}\text{-}Deep]$$

effect-free (wrt H_ϕ) after #

- Instantiate the high-order predicate k using the continuation's specification

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x, r) \qquad (E(y)k \to \Phi) \in \mathcal{H}_\Phi \qquad \Phi' = \Phi[x/y, \boxed{(\lambda(r, r_c) \to \Phi[r_c])/k]}}{\mathbf{try}[\delta](\mathcal{E}\ \#\ \Phi[r_c])\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{N}\ ;\ \Phi'}\ [\mathcal{R}\text{-}Eff\text{-}Handle]$$

Binding the effect free continuation to k

Higher-Order Function meets Unresolved Try-Catch Construct

```
let foo f : int
```
(* $foo(f, r) = \mathbf{try}\ (\exists res \cdot f(res))\ \ \mathbf{catch}\,\{\,Label\,k \rightarrow k(5, r)\,\}$ *)
```
= match f() with
| effect Label k -> resume k 5
```

Keep the try-catch constructs in the specification, which allows modular verification.

```
let goo () : int
```
(* $goo(r) \ = \ \exists x \cdot \mathbf{ens}[r]\ x {\mapsto} 0 \wedge r{=}15$ *)
```
= let f = (fun () -> let x = ref 0 in (perform Label) + 10)
  in foo f
```

Instantiate the unknown function

$$goo(r) \ = \exists f \cdot \ \mathbf{ens}\ f(res) = (\exists\, x, y \cdot \mathbf{ens}\ x \mapsto 0\,; Label(y); \mathbf{ens}[res]\ res{=}y{+}10)\ \ ;\ foo(f, r)$$

$$\rightsquigarrow \ \ \mathbf{try}\ \exists\, res, x, y \cdot \mathbf{ens}\ x \mapsto 0\,; Label(y); \mathbf{ens}[res]\ res{=}y{+}10\ \mathbf{catch}\,\{\,Label\,k \rightarrow k(5, r)\,\}$$

$$\rightsquigarrow^* \ \ \exists x \cdot \mathbf{ens}[r]\ x \mapsto 0 \wedge r{=}15$$

Reducing the try-catch construct

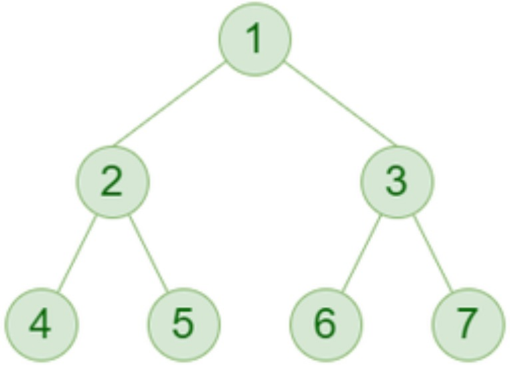
Inductive Proofs via Lemmas

```
effect Flip : bool

let tossN n
(* tossN(n, res) = ∃r0 ; ens n=1; Flip(r0); ens[res] res=r0 ∨
                   ∃r1 ; ens n>1; Flip(r1) ; ∃r2 ; tossN(n−1, r2) ; ens[res] res=(r1∧r2) *)
= match n with
| 1 -> perform Flip
| n -> let r1 = perform Flip in
       let r2 = tossN (n-1) in r1 && r2

let all_results counter n
(* all_results(n, r) = ∃z ; req counter ↦ z ∧ n>0 ens[r] counter ↦ z+(2^(n+1)−2) ∧ r=1 *)
= match tossN n with
  | x -> if x then 1 else 0
  | effect Flip k ->
      counter := !counter + 1;           (* increase the counter *)
      let res1 = resume k true in        (* resume with true     *)
      counter := !counter + 1;           (* increase the counter *)
      let res2 = resume k false in       (* resume with false    *)
      res1 + res2                        (* gather the results   *)
```

Conjunct each Flip result



n=1, counter = 2, res = 1

n=2, counter = 6, res =1

n=3, counter=14, res =1

…

… , counter = 2^{n+1}-2, res=1

Inductive Proofs via Lemmas

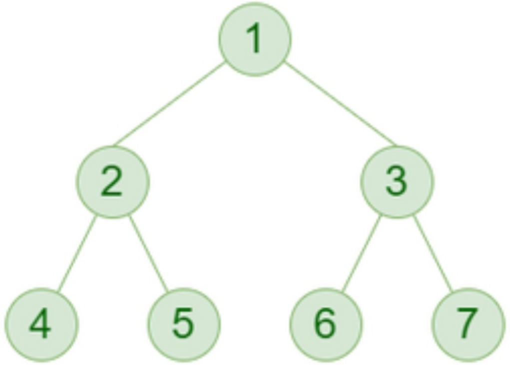
```
effect Flip : bool

let tossN n
(* tossN(n, res) = ∃r0 ; ens n=1; Flip(r0); ens[res] res=r0 ∨
                   ∃r1 ; ens n>1; Flip(r1); ∃r2; tossN(n−1, r2); ens[res] res=(r1∧r2) *)
= match n with
| 1 -> perform Flip
| n -> let r1 = perform Flip in
       let r2 = tossN (n-1) in r1 && r2

let all_results counter n
(* all_results(n, r) = ∃z ; req counter ↦ z ∧ n>0 ens[r] counter ↦ z+(2^(n+1)−2) ∧ r=1 *)
= match tossN n with
  | x -> if x then 1 else 0
  | effect Flip k ->
      counter := !counter + 1;         (* increase the counter *)
      let res1 = resume k true in      (* resume with true     *)
      counter := !counter + 1;         (* increase the counter *)
      let res2 = resume k false in     (* resume with false    *)
      res1 + res2                      (* gather the results   *)
```

Conjunct each Flip result

Sum up how many back tracking branches leads to all true



n=1, counter = 2, res = 1

n=2, counter = 6, res =1

n=3, counter=14, res =1

...

... , counter = 2^{n+1}-2, res=1

Inductive Proofs via Lemmas

```
effect Flip : bool
```

$$\textbf{try}\ \exists res\,;\ tossN(n, res)\ \#\ \exists r;\ \textbf{ens}[r]\ (acc{\wedge}res){\wedge}r{=}1 \vee \neg(acc{\wedge}res){\wedge}r{=}0\ \ \textbf{catch}\,\mathcal{H}_{\Phi}\ \sqsubseteq \exists r\,;\ \Phi_{inv}(n, acc, r)$$

$$\Phi_{inv}(n, acc, r)\ =\ \exists w\,;\ \textbf{req}\ counter \mapsto w\ \textbf{ens}[r]\ counter{\mapsto}w{+}(2^{n+1}{-}2) \wedge (acc{\wedge}r{=}1 \vee \neg acc{\wedge}r{=}0)$$

```
  | n -> let r1 = perform Flip in
         let r2 = tossN (n-1) in r1 && r2

let all_results counter n
(* all_results(n, r) = ∃z ; req counter ↦ z ∧ n>0 ens[r] counter ↦ z+(2^(n+1)-2) ∧ r=1 *)
= match tossN n with
  | x -> if x then 1 else 0
  | effect Flip k ->
      counter := !counter + 1;           (* increase the counter *)
      let res1 = resume k true in        (* resume with true     *)
      counter := !counter + 1;           (* increase the counter *)
      let res2 = resume k false in       (* resume with false    *)
      res1 + res2                        (* gather the results   *)
```

Inductive Proofs via Lemmas

```
effect Flip : bool
```

$$\textbf{try}\ \exists res\,;\ tossN(n, res)\ \#\ \exists r;\ \textbf{ens}[r]\ (acc \wedge res) \wedge r{=}1 \vee \neg(acc \wedge res) \wedge r{=}0\ \ \textbf{catch}\ \mathcal{H}_{\Phi}\ \sqsubseteq \exists r\,;\ \Phi_{inv}(n, acc, r)$$

$$\Phi_{inv}(n, acc, r)\ =\ \exists w\,;\ \textbf{req}\ counter \mapsto w\ \textbf{ens}[r]\ counter {\mapsto} w{+}(2^{n+1}{-}2) \wedge (acc \wedge r{=}1 \vee \neg acc \wedge r{=}0)$$

```
  | n -> let r1 = perform Flip in
         let r2 = tossN (n-1) in r1 && r2

let all_results counter n
(* all_results(n, r) = ∃z ; req counter ↦ z ∧ n>0 ens[r] counter ↦ z+(2^(n+1)-2) ∧ r=1 *)
= match tossN n with
  | x -> if x then 1 else 0
  | effect Flip k ->
      counter := !counter + 1;
      let res1 = resume k true in
      counter := !counter + 1;
      let res2 = resume k false in
      res1 + res2
```

- Proving via applying lemmas
- Lemmas are proved based on:
  - ✓ Try-catch reduction
  - ✓ Unfolding and rewriting (entailment rules)

Implementation and Evaluation

- 5K LoC on top of OCaml 5
- Benchmark programs with features: (Ind) proof is inductive, (MultiS)multi-shot handlers, (ImpureC) impure continuations, (HO) program is higher-order.

#	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

LoS/LoC < 30%

Summary

- ✓ **Scope**: Zero/one/multi-shots + impure continuations, deep/shallow handlers, left/right recursion
- ✓ **Effectful Specification Logic**: Staged specifications + unhandled effects + try-catch logic constructs
- ✓ **Hoare-style Verifier**: ML-like language + imperative higher-order + algebraic effects.
- ✓ **The Back-end Checker for ESL**: Normalization rules + reduction process of try-catch constructs.
- ✓ **Prototype (Multicore OCaml):** Proven correctness, report on experimental results, and case studies.

Summary

- ✓ **Scope**: Zero/one/multi-shots + impure continuations, deep/shallow handlers, left/right recursion
- ✓ **Effectful Specification Logic**: Staged specifications + unhandled effects + try-catch logic constructs
- ✓ **Hoare-style Verifier**: ML-like language + imperative higher-order + algebraic effects.
- ✓ **The Back-end Checker for ESL**: Normalization rules + reduction process of try-catch constructs.
- ✓ **Prototype (Multicore OCaml): ** Proven correctness, report on experimental results, and case studies.

Take Away:

Thanks!

1) **Try not to assume,** for both HO functions and effects!
2) Modular specifications without global assumptions: try-catch constructs.
3) Explicit access to the continuations, which can be composed as needed.

My Research

- PhD (2018 Aug – 2023 May)

Thesis: Symbolic Temporal Verification Techniques with Extended Regular Expressions

Keywords: Modularly (Scalability), Expressive Specification, Hoare-style Verification

Applications
- Event-based reactive systems [ICFEM 2020]
- Synchronous languages like Esterel [VMCAI 2021]
- User-defined algebraic effects and handlers [APLAS 2022]
- Real-time systems [TACAS 2023]

- Research Fellow (2023 – now)
  - Temporal Property Guided Bug Detection and Repair [FSE 2024]
  - Staged Specification Logic [FM 2024] [ICFP 2024]

Future

- PhD (2018 Aug – 2023 May)

  Thesis: Symbolic Temporal Verification Techniques with Extended Regular Expressions

  Keywords: Modularly (Scalability), Expressive Specification, Hoare-style Verification

  Applications:
  - Event-based reactive systems [ICFEM 2020]
  - Synchronous languages like Esterel [VMCAI 2021]
  - User-defined algebraic effects and handlers [APLAS 2022]
  - Real-time systems [TACAS 2023]

- Research Fellow (2023 – now)
  - Temporal Property Guided Bug Detection and Repair [FSE 2024]
  - Staged Specification Logic [FM 2024] [ICFP 2024]

➢ Staged Specification Logic

1) Summary/Lemma Inference;
2) Non-terminating/liveness behaviours of effects handling;
3) Apply staged specification to delimited control operators, e.g., shift/reset;
4) Combine staged specification with type system.

➢ Temporal Logic Based Bug Finding and Repair

1) Least Fixpoint & Greatest Fixpoint defined analysis : CTL + Datalog;
2) Liveness Checking: Termination + Safety checking + Fairness Assumption;
3) Temporal Logic Augmented LLM.

➢ …

References

[ICFEM 2020] Yahui Song and Wei-Ngan Chin. Automated temporal verification of integrated dependent effects. In Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony, editors, Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, Singapore, Singapore, March 1-3, 2021, Proceedings, volume 12531 of Lecture Notes in Computer Science, pages 73–90. Springer, 2020. doi: 10.1007/978-3-030-63406-3_5. URL https://doi.org/10.1007/ 978-3-030-63406-3_5.

[VMCAI 2021] Yahui Song and Wei-Ngan Chin. A synchronous effects logic for temporal verification of pure esterel. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, Copenhagen, Denmark, January 17-19, 2021, Proceedings, volume 12597 of Lecture Notes in Computer Science, pages 417–440. Springer, 2021. doi: 10.1007/978-3-030-67067-2_19. URL https://doi.org/10.1007/ 978-3-030-67067-2_19.

[APLAS 2022] Yahui Song, Darius Foo, and Wei-Ngan Chin. Automated temporal verification for algebraic effects. In Ilya Sergey, editor, Programming Languages and Systems - 20th Asian Symposium, Auckland, New Zealand, December 5, 2022, Proceedings, volume 13658 of Lecture Notes in Computer Science, pages 88–109. Springer, 2022. doi: 10.1007/978-3-031-21037-2_5. URL https://doi.org/10.1007/ 978-3-031-21037-2_5.

References

[TACAS 2023] Yahui Song and Wei-Ngan Chin. Automated verification for real-time systems - via implicit clocks and an extended antimirov algorithm. In Sriram Sankaranarayanan and Natasha Sharygina, editors, Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I, volume 13993 of Lecture Notes in Computer Science, pages 569–587. Springer, 2023. doi: 10.1007/978-3-031-30823-9_29. URL https: //doi.org/10.1007/978-3-031-30823-9_29.

[FSE 2024] Yahui Song, Xiang Gao, Wenhua Li, Wei-Ngan Chin, and Abhik Roychoudhury. Provenfix: Temporal property-guided program repair. Proceedings of the ACM on Software Engineering, 1(FSE):226–248, 2024b.

[FM 2024] Darius Foo, Yahui Song, and Wei-Ngan Chin. Staged specifications for automated verification of higher-order imperative programs. CoRR, abs/2308.00988, 2023. doi: 10.48550/ARXIV.2308.00988. URL https://doi.org/10.48550/arXiv.2308. 00988.

[ICFP 2024] Yahui Song, Darius Foo, and Wei-Ngan Chin. Specification and verification for unrestricted algebraic effects and handling. Proc. ACM Program. Lang., 8(ICFP), aug 2024a. doi: 10.1145/3674656. URL https://doi.org/10.1145/3674656.

References

[POPL 2021] Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. Proc. ACM Program. Lang. 5, POPL, Article 33 (January 2021), 28 pages. https://doi.org/10.1145/3434314.

[PhD Thesis 2022] Paulo Emílio de Vilhena. Proof of Programs with Effect Handlers. Computation and Language [cs.CL]. Université Paris Cité, 2022. English. ⟨NNT : 2022UNIP7133⟩. ⟨tel-03891381v3⟩

[ICFP 2011 (CFML)] Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP'11). Association for Computing Machinery, New York, NY, USA, 418–430. https://doi.org/10.1145/2034773.2034828

[POPL 2006] Zhaozhong Ni and Zhong Shao. 2006. Certified assembly programming with embedded code pointers. In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06). Association for Computing Machinery, New York, NY, USA, 320–333. https://doi.org/10.1145/1111037.1111066