NUS
National University
of Singapore

ICFP
MILAN 2024

Specification and Verification for Unrestricted Algebraic Effects and Handling

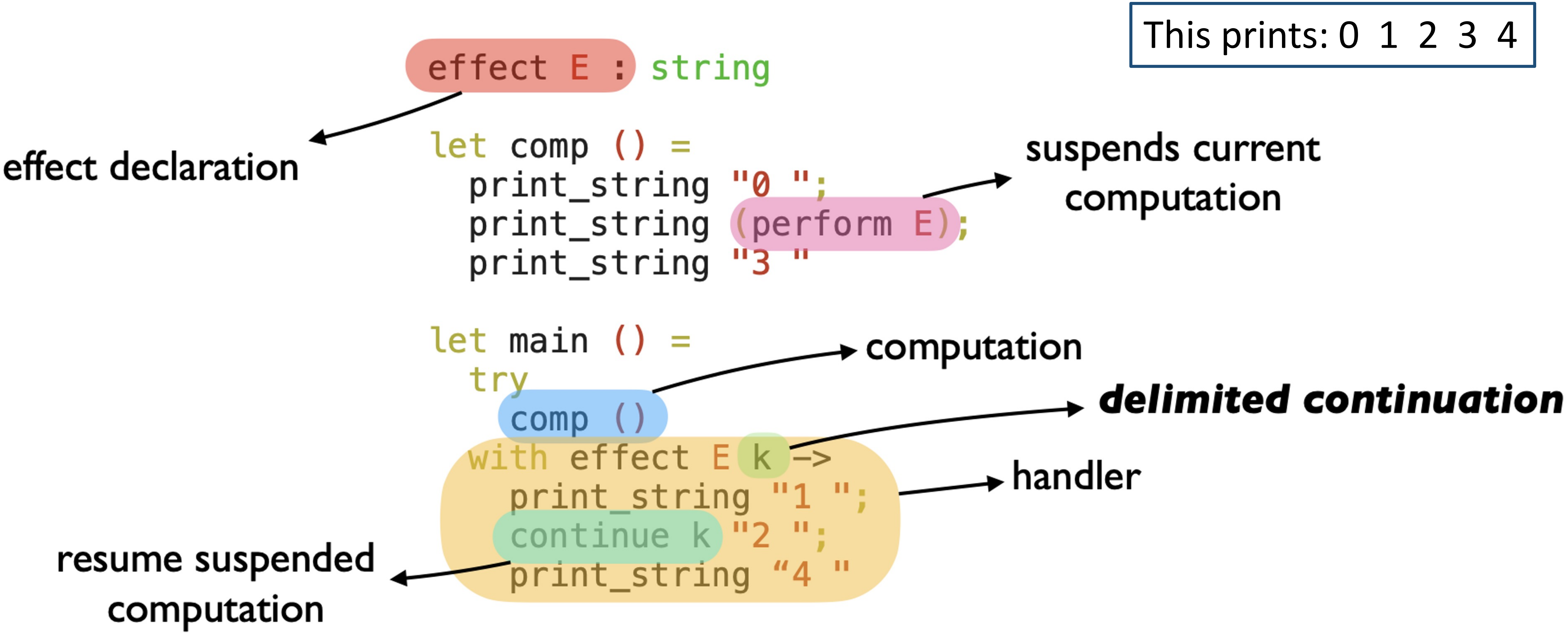
Artifacts Available acm V1.1

Artifacts Evaluated acm Functional V1.1

Yahui Song, Darius Foo, Wei-Ngan Chin

4th Sep @ ICFP 2024, Milan, Italy

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

Example taken from “Effect Handlers in Multicore OCaml” slides by KC Sivaramakrishnan.

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

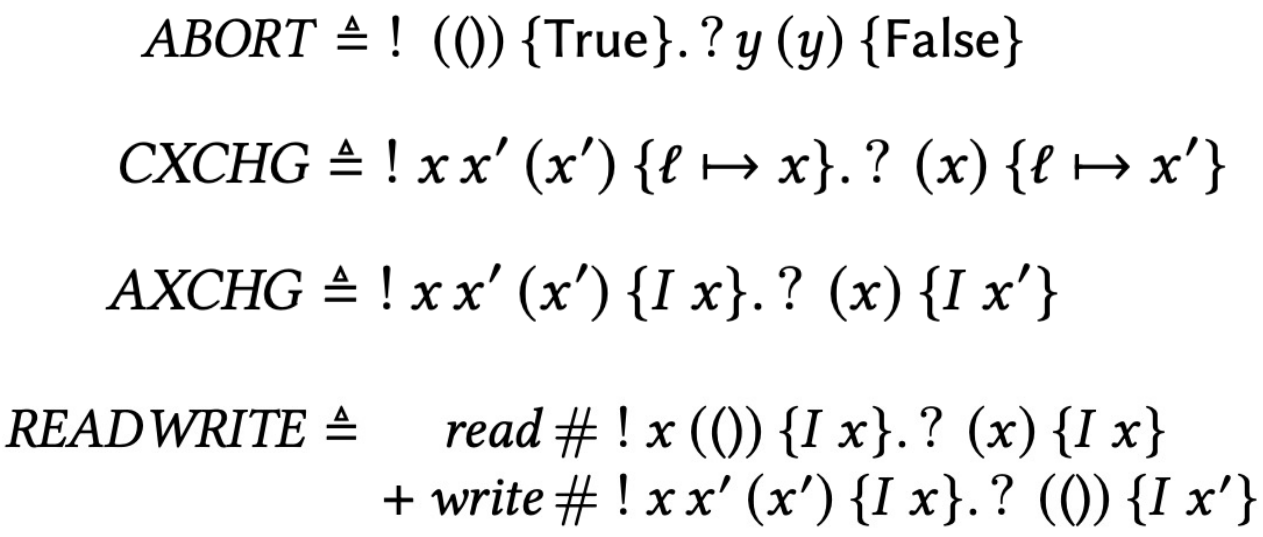
```
effect Label: int
(* User-defined effect, which will be resumed with int values *)

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



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

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

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

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

Our Solution: Effectful Specification Logic (ESL)

- Fully Modular Per-Method Verification (no global assumption)
- Sequencing, ***$\varphi_1\,;\,\varphi_2$***

result

- Uninterpreted relations for ***unhandled effects*** and unknown functions, $E(x, r)$

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

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

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

We propose ESL

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

```
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

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

We propose ESL

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

```
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:

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

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

Try-Catch Reduction (Examples)

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

```
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)

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

```
let zero_shot () : int
(* $\mathit{zero\_shot}(r_z) = \exists x\,;\, \mathbf{ens}[r_z]\; x{\mapsto}\mathit{0} \wedge r_z{=}{-}\mathit{1}$ *)
= match callee () with
| effect Label k -> -1
```

```
let one_shot () : int
(* $\mathit{one\_shot}(r_o) = \exists x\,;\, \mathbf{ens}[r_o]\; x{\mapsto}\mathit{1} \wedge r_o{=}\mathit{5}$ *)
= match callee () with
| effect Label k -> resume k 3
```

```
let multi_shot () : int
(* $\mathit{multi\_shot}(r_m) = \mathbf{req}\;\mathit{false}$ *)
= match callee () with
| effect Label k ->
  let _ = resume k 4 in resume k 5
```

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
```

```
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:

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

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

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

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

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

Try-Catch Reduction (Selected Rules)

- The base case:

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

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

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

effect-free (wrt H_ϕ) after #

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

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

Binding the effect free continuation to k

Higher-Order Function meets Unresolved Try-Catch Construct

```
let foo f : int
(* foo(f,r) = try (∃res · f(res))  catch { Label k → 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)  =  ∃x · ens[r] x↦0 ∧ r=15 *)
= let f = (fun () -> let x = ref 0 in (perform Label) + 10)
  in foo f
```

Instantiate the unknown function

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

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

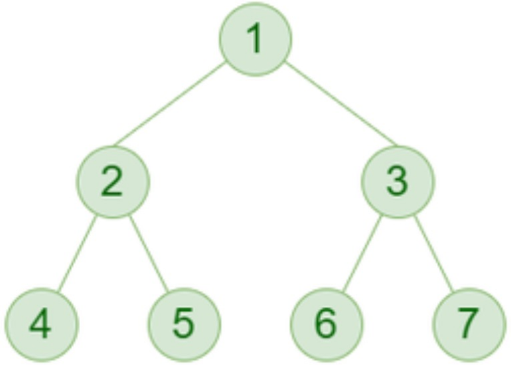
$$\rightsquigarrow^{*}\ \exists x \cdot \mathbf{ens}[r]\ x \mapsto 0 \wedge r{=}15$$

Reducing the try-catch construct

Inductive Proofs via Lemmas

```
effect Flip : bool

let tossN n
(* $tossN(n, res) = \exists r_0 ; \mathbf{ens}\ n{=}1; Flip(r_0); \mathbf{ens}[res]\ res{=}r_0 \lor$
                    $\exists r_1 ; \mathbf{ens}\ n{>}1; Flip(r_1); \exists r_2 ; tossN(n-1, r_2); \mathbf{ens}[res]\ res{=}(r_1 \land r_2)$ *)
= 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
(* $all\_results(n, r) = \exists z ; \mathbf{req}\ counter \mapsto z \land n{>}0\ \mathbf{ens}[r]\ counter \mapsto z{+}(2^{n+1}{-}2) \land r{=}1$ *)
= 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

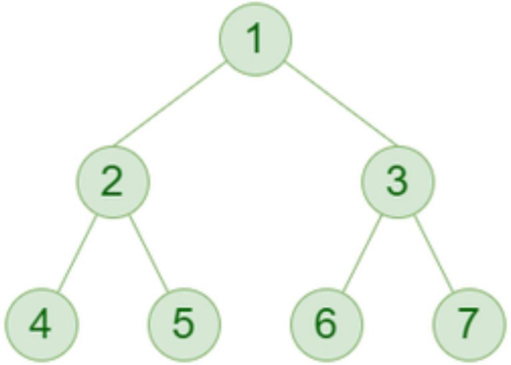
...

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

…

Inductive Proofs via Lemmas

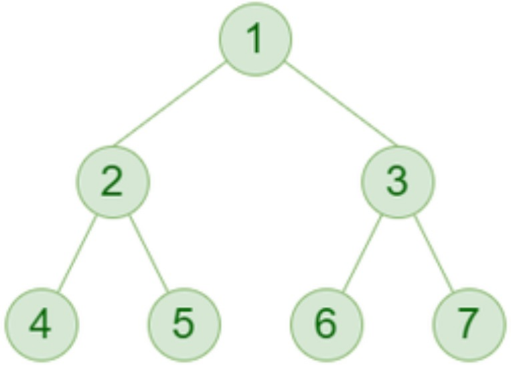
```
effect Flip : bool

let tossN n
```

$$(\ast\ \mathit{tossN}(n, \mathit{res}) = \exists r_0 ; \textbf{ens}\ \ n{=}1; \mathit{Flip}(r_0); \textbf{ens}[\mathit{res}]\ \mathit{res}{=}r_0\ \vee$$
$$\exists r_1 ; \textbf{ens}\ \ n{>}1 ; \mathit{Flip}(r_1) ; \exists r_2 ; \mathit{tossN}(n{-}1, r_2) ; \textbf{ens}[\mathit{res}]\ \mathit{res}{=}(r_1 {\wedge} r_2)\ \ \ast)$$

```
= 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
```

$$(\ast\ \mathit{all\_results}(n, r) = \exists z ; \textbf{req}\ \ \mathit{counter} \mapsto z \wedge n{>}0\ \ \textbf{ens}[r]\ \mathit{counter} \mapsto z{+}(2^{n+1}{-}2) \wedge r{=}1\ \ \ast)$$

```
= 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   *)
```



n=1, counter = 2, res = 1

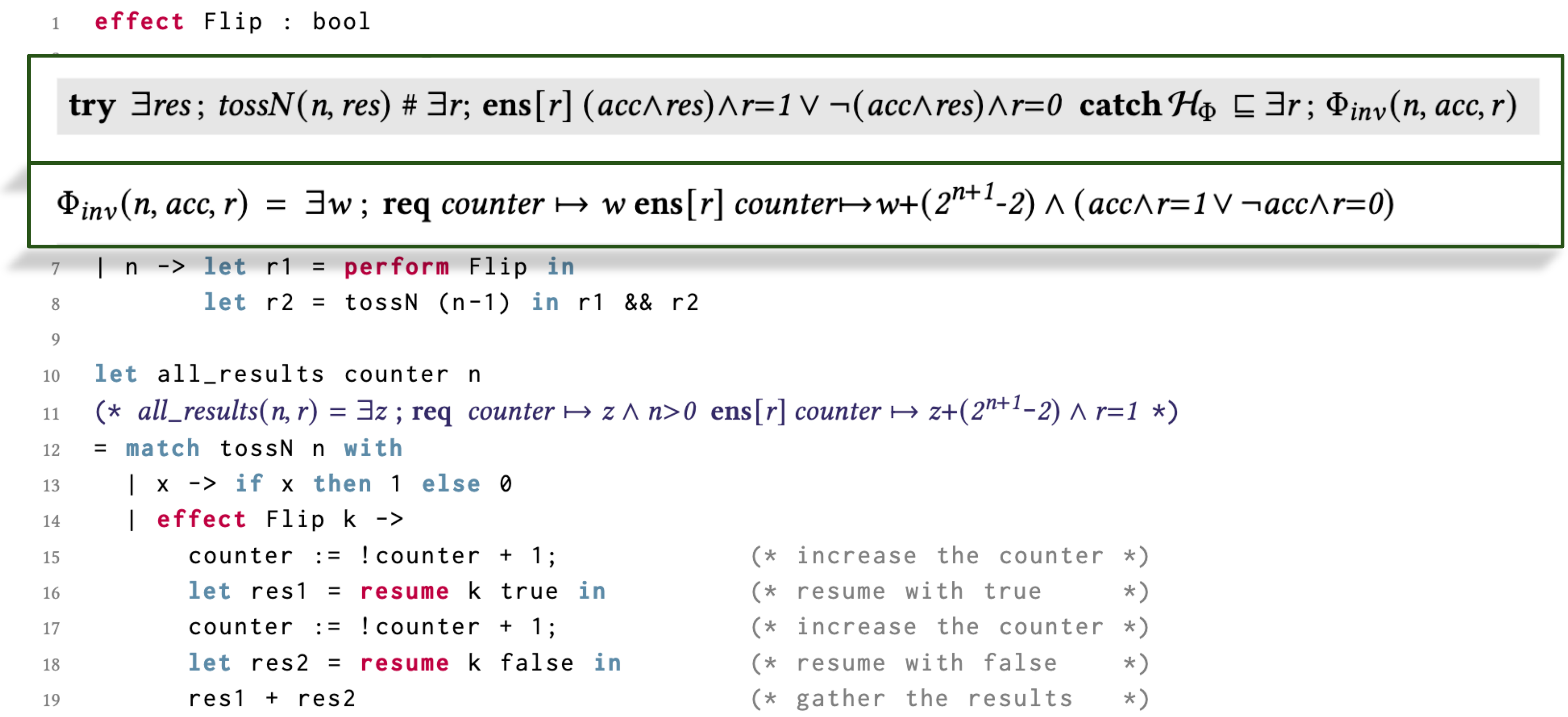
n=2, counter = 6, res =1

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
```

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

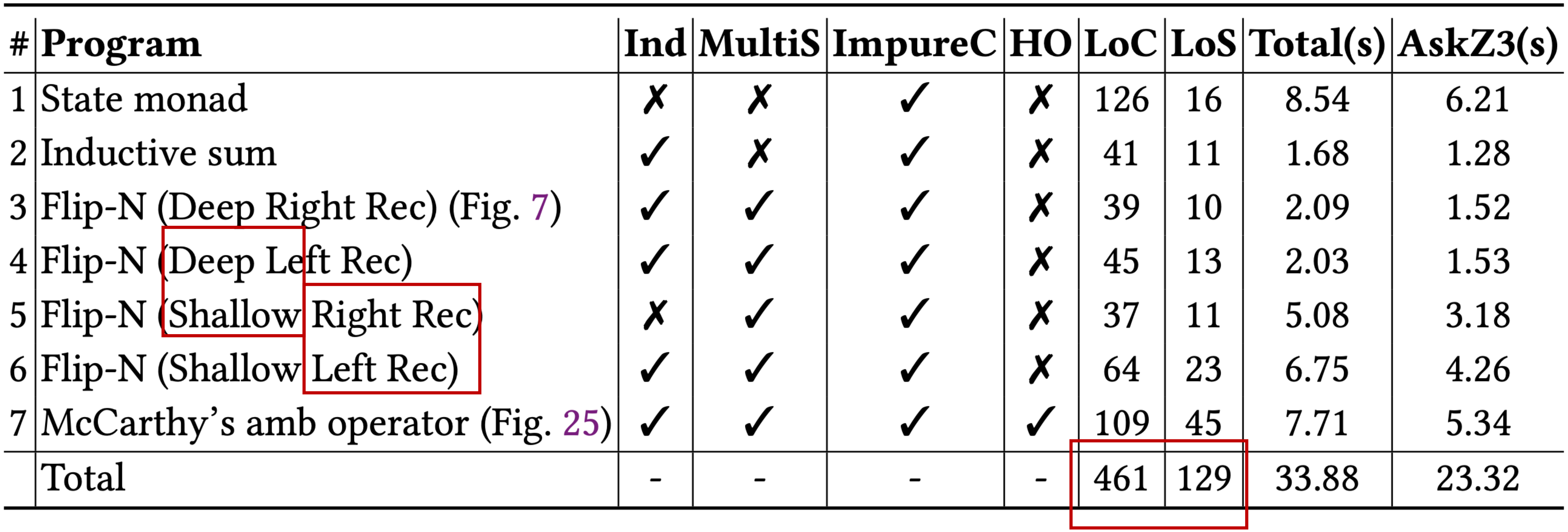
$$\Phi_{inv}(n, acc, r)\ =\ \exists w\,;\ \textbf{req}\ counter \mapsto w\ \textbf{ens}[r]\ counter{\mapsto}w{+}(2^{n+1}\text{-}2) \wedge (acc\wedge r{=}1 \vee \neg acc\wedge r{=}0)$$

```
  | 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:

1) **Try not to assume,** for both HO functions and effects!
2) Staged logic + try-catch enable modular specifications without global assumptions (protocols).
3) Explicit access to the continuations, which can be composed as needed.

I am currently on the academic job market, looking for research positions!

Thanks for listening!