Automated Temporal Verification for Algebraic Effects

Yahui Song, Darius Foo, Chin Wei Ngan

National University of Singapore

5th Dec 2022 @ APLAS 2022 in Auckland, New Zealand

Algebraic Effects

```
(* Spawn binary tree of tasks in depth-first order

        ************
         Fiber tree
        ************
              0
             /  \
            1    2
           / \  / \
          3   4 5  6
*)

let rec f id depth =
  if depth > 0 then begin
    fork (fun () -> f (id * 2 + 1) (depth - 1));
    fork (fun () -> f (id * 2 + 2) (depth - 1))
  end else begin
    yield ()
  end

let () = run (fun () -> f 0 2)
```

- User-defined *computational effects* and *handlers*
- Enables many language features (previously considered primitive) to be provided as libraries, usable in direct style!
  - Exceptions, generators, cooperative concurrency, asynchronous I/O, coroutines, nondeterminism

Example taken from https://github.com/ocaml-multicore/effects-examples

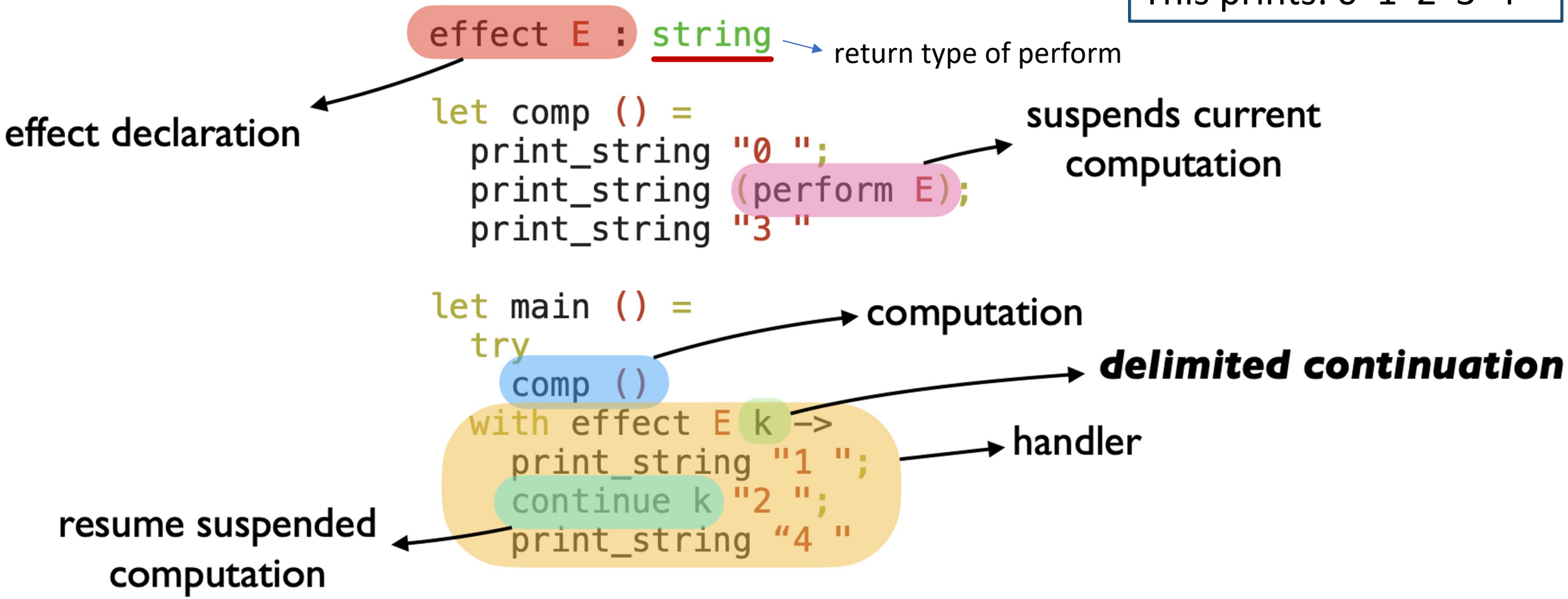
User-defined Effects and Handlers

```
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 "
```

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

User-defined Effects and Handlers

This prints: 0 1 2 3 4



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

Challenges

- Nonlocal control flow is hard to reason about
  - Can a given effect occur? Are all effects handled? ⟹ *effect system*

```
let main () =
  bar ();
  ... (* will this be executed? *)
```

```
fun sqr    : (int) -> total int        // total: mathematical total function
fun divide : (int,int) -> exn int      // exn: may raise an exception (partial)
fun turing : (tape) -> div int         // div: may not terminate (diverge)
fun print  : (string) -> console ()    // console: may write to the console
fun rand   : () -> ndet int            // ndet: non-deterministic
```

Examples taken from https://koka-lang.github.io/koka/doc/book.html#why-effects

Challenges

- Nonlocal control flow is hard to reason about
  - Can a given effect occur? Are all effects handled? ⟹ *effect system*
  - In what order are effects allowed to occur?

```
let main n =
  close_file n;
  open_file n
```

Challenges

- Nonlocal control flow is hard to reason about
  - Can a given effect occur? Are all effects handled? ⟹ *effect system*
  - In what order are effects allowed to occur?
  - Can the use of higher-order functions with deep handlers lead to nontermination?

```
1  effect Foo : (unit -> unit)
2
3  let f() = perform Foo ()
4
5  let loop()
6  = match f () with
7  | _ -> () (*normal return*)
8  | effect Foo k -> continue k
9    (fun () -> perform Foo ())
```

```
type 'a page = Page of 'a * (unit -> 'a page)

effect Request : int -> (string list) page

let database client =
  match client () with
  | () -> ()
  | effect (Request n) k ->
    let results = ... in
    continue k (Page (results,
      fun () -> perform (Request (n + size))))
```

```
let client () =
  let Page (results, next) = get 10 in
  ...
  let Page (results, next) = next () in
  ...
```

Challenges

- Nonlocal control flow is hard to reason about
- Multishot continuations are hard to use correctly
  - Many interesting use cases: nondeterminism, memoization, probabilistic programming
  - Does not mix well with imperative code, resources, linear continuations
  - “A separation logic for effect handlers” (POPL 2021) – focuses on zero-/one-shot

```
effect Choose : bool
let choose () = perform Choose

let all_results m =
  match m () with
  | v -> [v]
  | effect Choose k ->
     (continue k true) @ (continue (Obj.clone_continuation k) false)
```

```
let main () =
  bar_multishot ();
  ... (* how many times will this be executed? *)
  ... (* if we don't know, we can't close files, mutate variables etc. *)
```

Challenges

- Nonlocal control flow is hard to reason about
- Multishot continuations are hard to use correctly
- Modularity
  - We would like specify programs modularly, e.g. at function boundaries
  - However, reasoning about an effectful program can only be *fully* done when its interpretation (*handler*) is known

```
let g () =
  if choose ()
  then print_string "1"
  else print_string "2"
(* could print any of: nothing, 1, 2, 12, 112, 121, ... *)
```

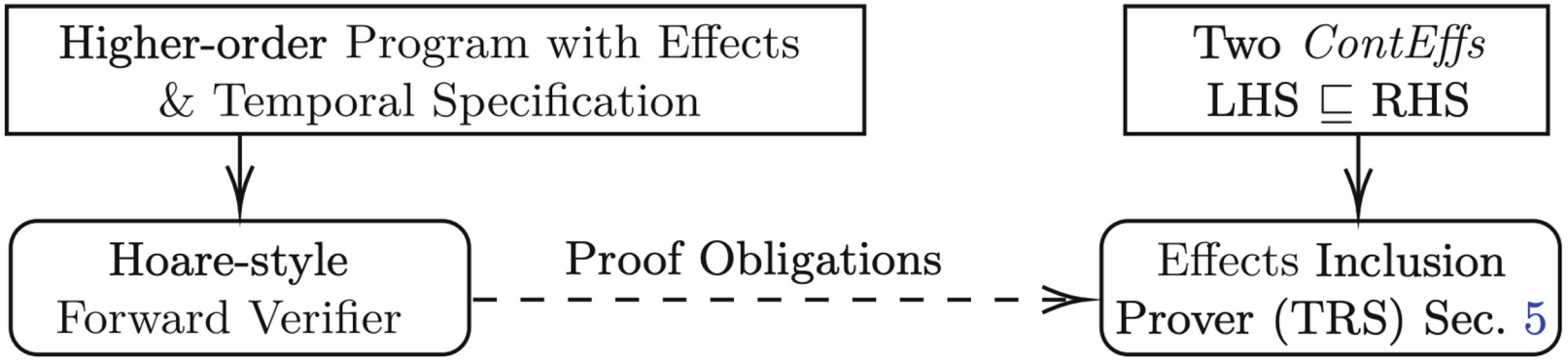
Challenges

1. Modularity
2. Multishot continuations
3. Nonlocal control flow

Contributions

1. A program logic with compositional temporal specifications, which handler reasoning uses
2. Coexistence of zero-shot, one-shot and multi-shot continuations
3. Fixpoint reasoning for some cases of deep handler nontermination

Verification Overview



- Specification Language ContEffs for sets of allowed traces
- Hoare-style forward verification, targeting an ML-like core language λ_h
  - Compositionally infers temporal behaviors of program via a set of forward rules
  - Fixpoint calculator to check for potential nontermination
- Term Rewriting System (TRS) checks entailments between ContEffs
- We prove soundness of the forward verifier and termination of the TRS

Core Language λ_h: pure, higher-order, call by value

$$
\begin{array}{ll}
(\mathit{Values}) & v ::= c \mid x \mid \lambda x \Rightarrow e \\
(\mathit{Expressions}) & e ::= v \mid v_1\ v_2 \mid \mathit{let}\ x{=}v\ \mathit{in}\ e \mid \mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \\
 & \quad \mathit{perform}\ A(v, \lambda x \Rightarrow e) \mid \mathit{match}\ e\ \mathit{with}\ h \mid \mathit{resume}\ v
\end{array}
$$

Specification Language ContEffs:

$$
\begin{array}{rrcl}
(\mathit{ContEffs}) & \Phi & ::= & \bigvee(\pi, \theta, v) \\
(\mathit{Parameterized\ Label}) & l & ::= & \Sigma(v) \\
(\mathit{Event\ Sequences}) & \theta & ::= & \bot \mid \epsilon \mid ev \mid Q \mid \theta_1 \cdot \theta_2 \mid \theta_1 \vee \theta_2 \mid \theta^\star \mid \theta^\infty \mid \theta^\omega \\
(\mathit{Single\ Events}) & ev & ::= & \_ \mid l \mid \bar{l} \\
(\mathit{Placeholders}) & Q & ::= & l! \mid l?(v)
\end{array}
$$

Motivating Example

```
1  effect Open : int -> unit
2  effect Close: int -> unit
3
4  let open_file n
5  (*@ req _^* @*)
6  (*@ ens Open(n)! @*)
7  = perform (Open n)
8
9  let close_file n
10 (*@ req _^*.Open(n)!.(~Close(n)!)^*  @*)
11 (*@ ens Close(n)! @*)
12 = perform (Close n)
```

```
13
14 let file_9 ()
15 (*@ req emp @*)
16 (*@ ens Open(9)!.Close(9)!.Close(9)! @*)
17 = open_file 9;
18   close_file 9;
19   close_file 9;
20
21 let main
22 (*@ req emp @*)
23 (*@ ens Open(9).(~Close(9))^*.Close(9) @*)
24 = match file_9 () with
25 | _ -> ()
26 | effect (Open n) k -> continue k ()
27 | effect (Close n) k -> continue k ()
```

Examples – One-shot continuations

```
1  effect Foo : (unit -> unit)
2  effect Goo : (unit -> unit)
3
4  let f ()
5  (*@ req emp    @*)
6  (*@ ens Foo!.Goo!.Goo?().Foo?() @*)
7  = let x = perform Foo in
8    let y = perform Goo i
9    y ();
10   x ()
11
12 let handler
13 (*@ req emp    @*)
14 (*@ ens Foo.Goo @*)
15 = match f () with
16 | x -> x
17 | effect Foo k -> continue k (fun () -> ())
18 | effect Goo k -> continue k (fun () -> ())
```

Step	History	Current Event	Continuation	Bindings
1	emp	Foo!	Goo! · Goo?() · Foo?() · ♥	♥ = (fun x -> x)
2	Foo	Goo!	Goo?() · Foo?() · ♥	Foo? = (fun () -> ())
3	Foo · Goo	Goo?()	Foo?() · ♥	Goo? = (fun () -> ())
4	Foo · Goo · emp	Foo?()	♥	
5	Foo · Goo · emp · emp	♥	-	
Final	Foo · Goo	-	-	

Examples – Zero-shot continuations (Exceptions)

```
1  effect Exc : (unit -> unit)
2  effect Other : (unit -> unit)
3
4  let f ()
5  (*@ req emp @*)
6  (*@ ens Exc!.Other!.Other?().Exc?() @*)
7  = let x = perform Exc in
8    let y = perform Other in
9    y ();
10   x ()
11
12 let handler
13 (*@ req emp @*)
14 (*@ ens Exc @*)
15 = match f () with
16 | x -> x
17 | effect Exc k -> ()
```

Step	History	Current Event	Continuation	Bindings
1	emp	Exc!	Other! · Other?() · Exc?() · ♥	♥ = (fun x -> x)
2	Exc	-	-	No “Continue”
Final	Exc	-	-	

Examples – Multi-shot continuation

```
1  effect Foo : (unit -> int)
2  effect Goo : (unit -> int)
3  effect Done: (unit)
4
5  let f ()
6  (*@ req emp @*)
7  (*@ ens Foo!.Goo!.Goo?().Foo?() @*)
8  = let x = perform Foo in
9    let y = perform Goo in
10   y (); x ()
11
12 let handler
13 (*@ req emp @*)
14 (*@ ens Foo.Goo.Done!.
15           Goo.Done! @*)
16 = match f () with
17 | x ->  perform Done;
18 | effect Foo k -> continue k (fun () -> ());
19                   continue k (fun () -> ())
20 | effect Goo k -> continue k (fun () -> ())
```

Examples – Multi-shot continuation

Step	History	Current Event	Continuation	Bindings
1	emp	Foo!	Goo! · Goo?() · Foo?() · ♥	♥ = (fun x -> perform Done)
2	Foo	Goo!	Goo?() · Foo?() · ♥ · Goo! · Goo?() · Foo?() · ♥	Foo? = (fun () -> ())
3	Foo · Goo	Goo?()	Foo?() · ♥ · Goo! · Goo?() · Foo?() · ♥	Goo? = (fun () -> ())
4	Foo · Goo	Foo?()	♥ · Goo! · Goo?() · Foo?() · ♥	
5	Foo · Goo	♥	Goo! · Goo?() · Foo?() · ♥	
6	Foo · Goo · Done!	Goo!	Goo?() · Foo?() · ♥	
7	Foo · Goo · Done! · Goo	Goo?()	Foo?() · ♥	
8	Foo · Goo · Done! · Goo · emp	Foo?()	♥	
9	Foo · Goo · Done! · Goo · emp · emp	♥	-	
10	Foo · Goo · Done! · Goo · emp · emp · Done!	-	-	
Final	Foo · Goo · Done! · Goo · Done!	-	-	

Examples – Non-terminating Fixpoint

```
1  effect Foo : (unit -> unit)
2  effect Goo : (unit -> unit)
3
4  let f ()
5  (*@ req emp    @*)
6  (*@ ens Foo!.Foo?() @*)
7  = let x = perform Foo in
8    x ()
9
10 let handler
11 (*@ req emp   @*)
12 (*@ ens Foo^w @*)
13 = match f () with
14 | x -> x
15 | effect Foo k ->
16   continue k (fun () -> perform Foo ())
17 | effect Goo k -> ()
```

Step	History	Current Event	Continuation	Bindings
1	emp	Foo!	Foo?() · ♥	♥ = (fun x -> x)
2	Foo	Foo?()	♥	Foo? = (fun () -> perform Foo ())
			Foo! · Foo?() · ♥	
Final	Foo · Foo^w	-	-	

Reoccurrence!

Implementation and Evaluation

- Core implementation: ~ 2500 LOC in OCaml, on top of Multicore OCaml (4.12.0)
- Validation: manually annotated synthetic test cases marked with expected outputs

No.	LOC	Infer(ms)	#Prop(✓)	Avg-Prove(ms)	#Prop(✗)	Avg-Dis(ms)
1	32	14.128	5	7.7786	5	6.2852
2	48	14.307	5	7.969	5	6.5982
3	71	15.029	5	7.922	5	6.4344
4	98	14.889	5	18.457	5	7.9562
5	156	14.677	7	10.080	7	4.819
6	197	15.471	7	8.3127	7	6.8101
7	240	18.798	7	18.559	7	7.468
8	285	20.406	7	23.3934	7	9.9086
9	343	26.514	9	16.5666	9	13.9667
10	401	26.893	9	18.3899	9	10.2169
11	583	49.931	14	17.203	15	10.4443
12	808	75.707	25	21.6795	24	16.9064

Scan Me For the Project Repository

Conclusion and Future Work

Thanks!

- New approach for verifying Algebraic Effects
  - Syntax and semantics of ContEffs
  - Automated Hoare-style forward verification + fixpoint computation
- Prototype system: experimental results and case studies
  - Modular temporal specifications
  - Coexistence of zero-shot, one-shot and multi-shot continuations
  - Detecting nontermination due to deep handlers + higher-order
  - **TODO**: Extend the ContEffs logic with mutable heap states

Scan Me For the Project Repository