Specification and Verification for Unrestricted Algebraic Effects and Handling

YAHUI SONG, School of Computing, National University of Singapore, Singapore
DARIUS FOO, School of Computing, National University of Singapore, Singapore
WEI-NGAN CHIN, Department of Computer Science, National University of Singapore, Singapore

Programming with user-defined effects and effect handlers has many practical use cases involving imperative effects. Additionally, it is natural and powerful to use multi-shot effect handlers for non-deterministic or probabilistic programs that allow backtracking to compute a comprehensive outcome. Existing works for verifying effect handlers are restricted in one of two ways: permitting multi-shot continuations under pure setting or allowing heap manipulation for only one-shot continuations, due to the employed protocol mechanism to model interactions between invoked effects and their handlers.

This work proposes a novel calculus called *Effectful Specification Logic* (*ESL*) to support unrestricted effect handlers, where zero-/one-/multi-shot continuations can co-exist with imperative effects and higher-order constructs. *ESL* captures behaviors in stages, and provides precise models to support invoked effects, handlers and continuations. To show its feasibility, we prototype an automated verification system for this novel specification logic, prove its soundness, report on useful case studies, and present experimental results. With this proposal, we have provided an extended specification logic that is capable of modeling arbitrary imperative higher-order programs with algebraic effects and continuation-enabled handlers.

CCS Concepts: • **Theory of computation → Logic and verification**; **Program specifications**.

Additional Key Words and Phrases: Multi-shot Continuations, Separation Logic, Automated Verification, Effectful Specification Logic

ACM Reference Format:
Yahui Song, Darius Foo, and Wei-Ngan Chin. 2024. Specification and Verification for Unrestricted Algebraic Effects and Handling. *Proc. ACM Program. Lang.* 8, ICFP, Article 267 (August 2024), 29 pages. https://doi.org/10.1145/3674656

1 INTRODUCTION

User-defined effects and effect handlers are a modular approach for delimited control. They offer the ability to suspend and resume computations, allowing information to be transmitted both ways. More specifically, an effect handler resembles an exception handler, i.e., control is transferred to an enclosing handler when performing an effect. Unlike exception handlers, each effect handler has access to its delimited continuation. By invoking this continuation, the handler can communicate a reply to the suspended computation before resuming its execution.

Designs for effect handler implementations [Bauer and Pretnar 2015; Leijen 2014; Phipps-Costin et al. 2023; Sivaramakrishnan et al. 2021], applications [Kawahara and Kameyama 2020; Leijen 2017; Nguyen et al. 2022], and verification solutions [de Vilhena and Pottier 2021; Song et al. 2022; Timany and Birkedal 2019] diverge upon whether it should be permitted or forbidden to invoke a

Authors' addresses: Yahui Song, School of Computing, National University of Singapore, Singapore, yahuis@comp.nus.edu.sg; Darius Foo, School of Computing, National University of Singapore, Singapore, dariusf@comp.nus.edu.sg; Wei-Ngan Chin, Department of Computer Science, National University of Singapore, Singapore, chinwn@comp.nus.edu.sg.

ACM 2475-1421/2024/8-ART267
https://doi.org/10.1145/3674656

```
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 *)
```

Fig. 1. Introductory Example, adapted from de Vilhena and Pottier [2021].

captured continuation more than once. Existing works for verifying effect handlers with resources fall into one of two categories: deal with multi-shot continuations only in a pure setting, or reason about heap-manipulating behaviors for exclusively one-shot continuations. For example, a recently proposed separation logic, Hazel [de Vilhena and Pottier 2021], takes the latter approach, while its extension, Maze [de Vilhena 2022], chooses the former, with a restriction that multiple resumptions cannot involve *non-persistent* assertions, i.e., heap operations or assertions containing resources.

A conclusion from prior work was *"With the traditional separation logic, allowing continuations to be invoked more than once breaks certain fundamental laws of reasoning about programs"* [de Vilhena and Pottier 2021]. In short, if a continuation can be resumed twice, then a code block can be entered once and exited twice, which was regarded as problematic, as illustrated by an OCaml example in Fig. 1. The program defines an effect *Label* of integer type, indicating that when resumed, its handler should provide an integer value. Function *callee* initializes a pointer x with 0 and performs *Label* on line 6. The code after line 6 essentially forms the "*continuation of performing Label*". Finally, the program returns the resumed value *ret* plus 2. So far, nothing is known about the handler for *Label*, and we observe different behaviors of *callee* depending on the specific handler:

- Zero-shot handlers abandon the continuation completely, just like exception handlers;
- One-shot handlers resume the continuation once, and the assertion on line 8 must succeed;
- Multi-shot handlers resume the continuation more than once, so x could be incremented multiple times; thus, the assertion on line 8 would fail for all but first invocation of the continuation.

To reason about such programs, we propose a novel *Effectful Specification Logic* (*ESL*) that offers new logic constructs for both *effect invocations* and their *unrestricted handlers*, where imperative effects and zero-/one-/multi-shot continuations can co-exist. Our effectful specifications extend pre/post specifications by explicitly supporting: effects as *uninterpreted relations*, *try−catch* handlers as reducible logic constructs, delimited continuations as *lambda-bound relations*; and interspersing these with pre/post summaries that may appear before and/or after these logic constructs.

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

Fig. 2. *ESL* Specification for *callee*.

A precise *ESL* specification for *callee* is shown in Fig. 2 using three stages that are separated by logical construct ';' to capture computational sequencing. The first stage for Line 5 captures the spec of code fragment (`let x = ref 0 in` [.]) where [.] denotes the next evaluation context. We

shall use the notation *code* :: *sp* to associate each code fragment with its specification. Thus:

$$(\texttt{let x = ref 0 in [.]}) \;::\; \exists x \cdot \mathbf{ens}\ x{\mapsto}0$$

Here, the specification at this stage indicates that a heap memory location (represented by a points-to $x{\mapsto}0$ of separation logic [Calcagno et al. 2009]) will be created by the code fragment. Also, local variables are existentially quantified using $\exists\, x \cdot P$ whose scope may extend past P to the end of method's specification. Line 6 captures the invoked effect *Label*, as denoted by:

$$(\texttt{let ret = perform Label in [.]}) \;::\; \exists ret \cdot \mathit{Label}(ret)$$

Here, $\mathit{Label}(ret)$ denotes an algebraic effect invocation. Each algebraic effect is denoted by an uninterpreted relation $E(v^*, r)$, with arguments v^* and result r. The interpretation of such effects would come from concrete handlers that catches these effects (elaborated later in Sec. 2).

Lines 7-9 summarize the continuation code after *Label* via a pair of pre/post with assertion on Line 8 captured by the precondition ($z{+}1{=}1$). Using $[r_c]$ to capture the result of some post-state, we can thus model this last specification stage precisely using:

$$(\texttt{x:=!x+1; assert (!x=1); ret+2}) \;::\; \exists z \cdot \mathbf{req}\ x{\mapsto}z \wedge z{+}1{=}1\ \ \mathbf{ens}[r_c]\ x{\mapsto}z{+}1 \wedge r_c{=}(ret{+}2)$$

It is also possible to use over-approximation to obtain simpler simplifications, where helpful. For example, if the outcome of *callee* need not be tracked, we can over-approximate the last stage with:

$$(\texttt{x:=!x+1; assert (!x=1); ret+2}) \;::\; \exists z \cdot \mathbf{req}\ x{\mapsto}z \wedge z{+}1{=}1\ \ \mathbf{ens}[r_c]\ x{\mapsto}z{+}1$$

ESL specification is generally more complex than say two-stage pre/post specifications since it allows us to model program codes with algebraic effects more precisely via multiple stages. One key benefit of *ESL* is its ability to delay the interpretation for algebraic effect to callers' sites where try-catch handlers' specifications and scope of continuation become known.

2 MAIN CONTRIBUTIONS

With $callee(r_c)$ defined, we next write *ESL* specifications for different and representative (zero-/one-/multi-shot) handlers, in Fig. 3, Fig. 4, and Fig. 5, respectively. For simplicity, when handlers' normal-return clauses are identity functions, i.e., (| x -> x), we omit them.

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

Fig. 3. A Zero-Shot Handler with its Spec.

To model effect handlers precisely, we employ a new logic construct: $\mathbf{try}\ (\Phi)\ \mathbf{catch}\ \{pat_i{\rightarrow}\Phi_i\}_{i=1}^{n}$, as a logical counterpart of the *match-with* statement – where Φ is the specification of the try block, and each pair $\{pat_i{\rightarrow}\Phi_i\}$ denotes the specification of each handling case. Our algebraic effect handling constructs are always verified modularly, in that each handler declaration is only verified once, and each effect invocation can always be replaced by its already verified handling logic. For convenience, we shall use a context notation $\Phi[r]$ where r explictly identifies the final result of specification Φ. Also, $\Phi[\_]$ is a shorthand for $(\exists r \cdot \Phi[r])$. As an example, the specification for *zero_shot* could be initially modeled as shown below before it is reduced to its counterpart without the try-catch logic construct:

$$\begin{array}{rcl} \mathit{zero\_shot}(r_z) & = & \mathbf{try}\ (\exists r \cdot \mathit{callee}(r))\ \mathbf{catch}\ \{\, \mathit{Label}\ k \rightarrow \mathbf{ens}[r_z]\ r_z{=}{-}1 \,\} \\ & \leadsto^* & \exists x \cdot \mathbf{ens}[r_z]\ x{\mapsto}0 \wedge r_z{=}{-}1 \end{array}$$

When suitably instantiated[1], each instance of a *try-catch* logical construct is reducible to a simpler specification without it. For $zero\_shot(r_z)$, it is reducible to just a pre/post specification, as

[1] An example that is not suitably instantiated will be highlighted later in Sec 3.3.

shown above. To reason about the behavior of a handler, we utilize the code specification ($callee(r)$ in this case) to obtain the stages up to the invoked effect that we are able to handle, namely:

$$\exists x, ret \cdot \mathbf{ens}\ \ x{\mapsto}0\ ;\ Label(ret)$$

and leave the remaining stages to be the specification for the continuation, binding it to k:

$$k\ \ =\ \ \lambda(inp, r_k){\rightarrow}\ \exists z \cdot \mathbf{req}\ \ x{\mapsto}z \wedge z{+}1{=}1\ \ \mathbf{ens}[r_k]\ x{\mapsto}z{+}1 \wedge r_k{=}(inp+2).$$

For *zero_shot*, the continuation was never invoked and its match handler simply returned $res{=}{-}1$ as specified, together with heap state $x{\mapsto}0$ constructed earlier. Returning to *one_shot* shown in Fig. 4, its specification can be constructed and reduced as follows:

$$\begin{array}{rcl} one\_shot(r_o) & = & \mathbf{try}\ callee(\_)\ \ \mathbf{catch}\ \{Label\ k{\longrightarrow}\ k(3, r_o)\} \\ & \rightsquigarrow^* & \exists x \cdot \mathbf{ens}[r_o]\ x{\mapsto}1 \wedge r_o{=}5. \end{array}$$

The continuation call (`resume k 3`) is initially modelled as an uninterpreted relation $k(3, r_o)$ that is subsequently interpreted by the handler that catches the *Label* effect. Here, we could simplify *one_shot*'s specification to just a pre/post specification, which results in the heap location x being updated to 1 after the precondition $\exists z \cdot \mathbf{req}\ \ x{\mapsto}z \wedge z{+}1{=}1$ is successfully checked.

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

Fig. 4. A One-Shot Handler with its Spec.

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

Fig. 5. A Multi-Shot Handler with its Spec.

Lastly, we construct the specification for the *multi_shot* function (in Fig. 5) as follows:

$$multi\_shot(r_m)\ \ =\ \ \mathbf{try}\ callee(\_)\ \ \mathbf{catch}\ \{Label\ k{\longrightarrow}\ k(4, \_)\ ;\ k(5, r_m)\} \rightsquigarrow^*\ \mathbf{req}\ \ false$$

The specification for *multi_shot* is reduced to a *false* precondition, because the second time that continuation resume would violate its precondition $\exists z \cdot \mathbf{req}\ \ x{\mapsto}z \wedge z{+}1{=}1$, since the heap state after the first time continuation invocation would be $x{\mapsto}1$. As a result, the only safe specification is **req** *false*, which forbids this function from ever being (safely) called. As an alternative scenario, the verification of *multi_shot* would have succeeded if the assertion in *callee* were weakened from $(!x{=}1)$ to $(!x{\geq}1)$. Under this weaker check, the *ESL* specification for *weak_callee* would have been:

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

With this change, our reasoning would simplify the specification for the *multi_shot* function to its expected outcome: $(\exists x \cdot \mathbf{ens}[r_m]\ x{\mapsto}2 \wedge r_m{=}7)$. This is possible since the preconditions can now be successfully checked for both continuation calls, $k(4, \_)$ and $k(5, r_m)$, with the heap location x incremented twice before the result $r_m{=}7$ of the second k call be successfully returned.

Unlike prior work [de Vilhena 2022; de Vilhena and Pottier 2021; Soares and Pereira 2023], which is unable to verify programs with *heap-based* continuation for multi-shot handlers, we have briefly shown here that sound reasoning is still achievable with the help of our new logic constructs for

effect invocations and try-catch handling. One inherent limitation of the previous solution is its reliance on a Player-Opponent protocol logic to model code leading to the continuation call in the handler as communication between programs and handlers via send/receive commands, and to use another sequence of ghost instructions in reverse-mode to model the code after the continuation call in the handler. This modeling supports heap mutation for only single-shot handlers, and it quickly becomes awkward when facing multi-shot handlers. In contrast, *ESL* captures behavior in stages, partitioned by invoked effects, which allows continuations to be captured and manipulated symbolically and in a delimited fashion. This allows us to model algebraic effects abstractly and provides a more precise mechanism for effect handlers without imposing restrictions on them.

Although simple, these examples show the capabilities of our proposal: i) *ESL* allows assertions to materialize as heap-based pre-conditions of captured continuations, lending itself to verifying heap-manipulating multi-shot continuations; ii) *ESL* naturally models unhandled effects as a relation between input(s) and an output, which is paired with a corresponding outer handler; and iii) we extend *automated* verification for multi-shot ceffect handlers and heap-manipulating continuations, which cannot be verified by the current state-of-the-art systems. Our contributions are:

(1) **Effectful Specification Logic:** We define the syntax and semantics of *ESL*, which captures staged specifications of heap-operations and assertions, explicitly revealing unhandled effects together with novel try-catch handlers that are usually reducible.
(2) **Hoare-style Verifier:** Targeting an ML-like language with both imperative higher-order features and algebraic effects, we establish forward rules to compositionally summarize and verify programs' behaviors. The verification utilizes a back-end entailment checker for *ESL*.
(3) **The Back-end Checker for *ESL*:** Our back-end checker proves/disproves the entailments between two normalized *ESL* formulae. We achieve this with the help of a set of normalization rules and a reduction process for try-catch logic constructs, where possible.
(4) **Implementation and Evaluation:** We prototype our proposed verifier, prove its correctness, report on experimental results, and present various case studies investigating *ESL*'s capabilities. Our target programs and our implementation are both written in Multicore OCaml.

3 ILLUSTRATIVE EXAMPLES

This section presents a few non-trivial examples to show the core idea and benefits of our approach.

3.1 Passing Pointers with an Effect Invocation

Consider the example in Fig. 6, which manipulates two pointers. Line 7 invokes effect E with pointer references i and j as arguments. Then by line 8, we have lost the information on the concrete values of i and j, because the handler could modify their contents. Although simple, this example shows why traditional pre/post specifications cannot handle such complex control mechanisms. Our proposed *ESL* resolves the issue by allowing a new stage after performing E(i,j,ret) which uses two existential variables x and y to denote the values of i and j, at the resumed point (Line 8). Moreover, this program may have multi-shot handlers,

```
1  effect E: (int ref * int ref) -> unit
2
3  let two_pointers ()
4  (* two_pointers(r) = ∃i,j,ret · ens i↦0 * j↦0 ; E(i,j,ret);
       ∃x,y · req i↦x * j↦y ens[r] i↦x+1 * j↦y+1∧r=() *)
5  = let i = ref 0 in
6    let j = ref 0 in
7    let ret = perform E (i, j) in
8    i := !i + 1;
9    j := !j + 1
```

Fig. 6. Two Pointers with an Effect Invocation.

and at each resumption, x and y will be replaced by fresh variables. From another perspective, *ESL* essentially takes "invoking effects" and "resuming continuations" as function calls to support modular verification via staged specifications. This paper shows how to soundly compose such specification stages during both forward reasoning and try-catch reduction.

3.2 Multi-Shot Handler with an Imperative Counter

```
1  effect Flip : bool
2
3  let tossN n
4  (* 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) *)
5  = match n with
6  | 1 -> perform Flip
7  | n -> let r1 = perform Flip in
8         let r2 = tossN (n-1) in r1 && r2
9
10 let all_results counter n
11 (* all_results(n, r) = ∃z·req counter ↦ z ∧ n>0 ens[r] counter ↦ z+(2^(n+1)-2) ∧ r=1 *)
12 = match tossN n with
13   | x -> if x then 1 else 0
14   | effect Flip k ->
15       counter := !counter + 1;            (* increase the counter *)
16       let res1 = resume k true in         (* resume with true     *)
17       counter := !counter + 1;            (* increase the counter *)
18       let res2 = resume k false in        (* resume with false    *)
19       res1 + res2                         (* gather the results   *)
```

Fig. 7. Flipping a Coin n-times.

Fig. 7 presents a multi-shot handler for a backtracking computation and increases a mutable *counter* whenever the continuation is resumed. Such uses of multi-shot continuations can be found in search problems [de Vilhena 2022], and simulation for probabilistic programs [Nguyen et al. 2022]. We now show that *ESL* is able to safely verify mutable states with multi-shot continuations.

The *tossN* function takes an argument n, invokes the effect *Flip* n times, and recursively computes a boolean conjunction of all the resumed results. When handling *Flip*, from line 15, the handler resumes the execution twice, with values *true* and *false* respectively, and before each resumption, it increases the counter by 1. As the specification for function *all_results* shows, given the input counter originally points to z and n greater than 0, our verifier proves that *counter* points to

$$\mathcal{H}_\Phi = \{x \rightarrow \exists r \cdot \mathbf{ens}[r]\ (x \wedge r{=}1) \vee (\neg x \wedge r{=}0)\,,$$
$$\mathit{Flip}()k \rightarrow \exists z_1, r_1 \cdot \mathbf{req}\ \mathit{counter} \mapsto z_1\ \mathbf{ens}\ \mathit{counter} \mapsto z_1{+}1\,;\ k(\mathit{true}, r_1)\,;$$
$$\exists z_2, r_2 \cdot \mathbf{req}\ \mathit{counter} \mapsto z_2\ \mathbf{ens}\ \mathit{counter} \mapsto z_2{+}1\,;\ k(\mathit{false}, r_2)\,;\ \mathbf{ens}[r]\ r{=}r_1{+}r_2\}$$

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

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

Fig. 8. A Try-Catch Lemma Deployed (highlighted in gray), and its Definition.

($z{+}2^{n+1}{-}2$) by the end of the execution and the return value is always 1. For example, taking $counter{\mapsto}0$ and $n{=}2$ as a concrete state when executing *all_results*, the counter will be indeed updated to 6, i.e., $0{+}2^{2+1}{-}2$, since the first *Flip* increments *counter* by two and explores both the *true* and *false* possibilities. For each of these two possibilities, a subsequent *Flip* in the continuation will each explore two more possibilities; hence, in total, *counter* will be increased six times. As for the return value, it represents the times when all the flips are true; hence, it is always 1.

To symbolically prove *all_results*'s specification, our verifier summarises the specifications for the handler cases using $\mathcal{H}_\Phi$. Due to the use of recursion, our verifier uses lemmas that could be inductively proven. For this example, we first specify the lemma in Fig. 8. We use the # operator to mean that the flows after # had already been handed by the handler, which is explained in detail in Sec. 5.2. This lemma captures the behavior of a try-catch construct using a one stage summary, i,e., $\Phi_{inv}(n, acc, r)$. In particular, r denotes the integer outcome returned by the normal clause, and the formula "$\#\ (\exists r \cdot \mathbf{ens}[r] \cdots)$" captures a continuation occurring after the $tossN(n, res)$ call.

$$\mathbf{try}\ \exists res \cdot tossN(n, res)\ \#\ \exists r \cdot\ \mathbf{ens}[r]\ (acc{\wedge}res){\wedge}r{=}1 \vee \neg(acc{\wedge}res){\wedge}r{=}0\ \mathbf{catch}\ \mathcal{H}_\Phi \quad \text{(When n=1)}$$

$$\rightsquigarrow \mathbf{try}\ \exists res \cdot\ \mathbf{ens}\ n{=}1;\ Flip(res)\ \#\ \exists r \cdot\ \mathbf{ens}[r]\ (acc{\wedge}res){\wedge}r{=}1 \vee \neg(acc{\wedge}res){\wedge}r{=}0\ \mathbf{catch}\ \mathcal{H}_\Phi \quad \boxed{[\mathcal{R}\text{-}Eff\text{-}Handle]}$$

$$\rightsquigarrow \mathbf{ens}\ n{=}1;\ \exists w_1 \cdot \mathbf{req}\ counter{\mapsto}w_1\ \mathbf{ens}\ counter{\mapsto}w_1{+}1\ ;\ \exists r'_1 \cdot \mathbf{ens}[r'_1]\ (acc{\wedge}true{\wedge}r'_1{=}1 \vee \neg(acc{\wedge}true){\wedge}r'_1{=}0);\ \exists w_2 \cdot\ \mathbf{req}\ counter{\mapsto}w_2\ \mathbf{ens}\ counter{\mapsto}w_2{+}1\ ;\ \exists r'_2 \cdot \mathbf{ens}[r'_2]\ (acc{\wedge}false{\wedge}r'_2{=}1 \vee \neg(acc{\wedge}false){\wedge}r'_2{=}0);\ \exists r \cdot \mathbf{ens}[r]\ r{=}r'_1{+}r'_2$$

$$\rightsquigarrow \exists r, w \cdot \mathbf{req}\ counter{\mapsto}w\ \mathbf{ens}[r]\ counter{\mapsto}w{+}2 \wedge n{=}1 \wedge (acc{\wedge}r{=}1 \vee \neg acc{\wedge}r{=}0) \quad \sqsubseteq \quad \exists r \cdot \Phi_{inv}(1, acc, r)$$

$$\mathbf{try}\ \exists res \cdot tossN(n, res)\ \#\ \exists r \cdot\ \mathbf{ens}[r]\ (acc{\wedge}res){\wedge}r{=}1 \vee \neg(acc{\wedge}res){\wedge}r{=}0\ \mathbf{catch}\ \mathcal{H}_\Phi \quad \text{(When n>1)}$$

$$\rightsquigarrow \mathbf{try}\ \exists r_1 \cdot\ \mathbf{ens}\ n{>}1;\ Flip(r_1)\ ;\ \exists r_2 \cdot tossN(n{-}1, r_2)\ \#\ \exists r \cdot\ \mathbf{ens}[r]\ (acc{\wedge}r_1{\wedge}r_2){\wedge}r{=}1 \vee \neg(acc{\wedge}r_1{\wedge}r_2){\wedge}r{=}0\ \mathbf{catch}\ \mathcal{H}_\Phi \quad \boxed{[\mathcal{R}\text{-}Lemma\text{-}App]}$$

$$\rightsquigarrow \mathbf{try}\ \exists r_1 \cdot\ \mathbf{ens}\ n{>}1;\ Flip(r_1)\ \#\ \exists r \cdot\ \Phi_{inv}(n{-}1, acc{\wedge}r_1, r)\ \mathbf{catch}\ \mathcal{H}_\Phi \quad \boxed{[\mathcal{R}\text{-}Eff\text{-}Handle]}$$

$$\rightsquigarrow \mathbf{ens}\ n{>}1;\ \exists w_1 \cdot \mathbf{req}\ counter{\mapsto}w_1\ \mathbf{ens}\ counter{\mapsto}w_1{+}1\ ;\ \exists r'_1 \cdot \Phi_{inv}(n{-}1, acc{\wedge}true, r'_1)\ ;\ \exists w_2 \cdot\ \mathbf{req}\ counter{\mapsto}w_2\ \mathbf{ens}\ counter{\mapsto}w_2{+}1\ ;\ \exists r'_2 \cdot \Phi_{inv}(n{-}1, acc{\wedge}false, r'_2)\ ;\ \exists r \cdot \mathbf{ens}[r]\ r{=}r'_1{+}r'_2$$

$$\rightsquigarrow \exists r, w \cdot \mathbf{req}\ counter{\mapsto}w\ \mathbf{ens}[r]\ counter{\mapsto}w{+}1{+}(2^n{-}2){+}1{+}(2^n{-}2)\ \wedge\ n{>}1 \wedge (acc{\wedge}r{=}1 \vee \neg acc{\wedge}r{=}0)$$

$$\rightsquigarrow \exists r, w \cdot \mathbf{req}\ counter{\mapsto}w\ \mathbf{ens}[r]\ counter{\mapsto}w{+}(2^{n+1}{-}2)\ \wedge\ n{>}1 \wedge (acc{\wedge}r{=}1 \vee \neg acc{\wedge}r{=}0) \quad \sqsubseteq \quad \exists r \cdot \Phi_{inv}(n, acc, r)$$

Fig. 9. Proving the Lemma in Fig. 8 (reduction rules are boxed), including the Base and Inductive cases.

Our lemma-proving process unfolds the recursive predicate $tossN(n, res)$, before showing that it can be proven to hold for both the base case (when n=1) and the inductive case (when n>1), shown in Fig. 9. Next, after the try-catch lemma has been proven, it can now be used by the try-catch reduction (cf. Sec. 5.2). As shown in Fig. 10, the rule $[\mathcal{R}\text{-}Lemma\text{-}App]$ reduces the formula into the instantiated (verified) lemma, i.e., $\Phi_{inv}(n, true, r)$. Finally, the entailment checking – denoted by $\sqsubseteq$ – succeeds, and the verification for the *all_results* function completes.

$$\mathbf{try}\ \exists res \cdot tossN(n, res)\ ;\mathbf{ens}[res]\ emp\ \ \mathbf{catch}\ \mathcal{H}_\Phi \quad \boxed{[\mathcal{R}\text{-}Deep]}$$

$$\rightsquigarrow \mathbf{try}\ \exists res \cdot tossN(n, res)\ \#\ \exists r \cdot \mathbf{ens}[r]\ (res{\wedge}r{=}1) \vee (\neg res{\wedge}r{=}0)\ \mathbf{catch}\ \mathcal{H}_\Phi \quad \boxed{[\mathcal{R}\text{-}Lemma\text{-}App]}$$

$$\rightsquigarrow \exists r \cdot \Phi_{inv}(n, true, r)\ \sqsubseteq\ \exists r \cdot all\_results(n, r) \quad \square$$

Fig. 10. Try-catch Reduction when Handling tossN, and the Entailment Generated for Function *all_results*.

Although this example is based on a deep handler and a right recursion, we show that our verification approach with the usage of lemmas can cover other non-trivial cases, such as a deep handler with a left recursion, or a shallow handler with both the right and left recursion, respectively. We demonstrate such examples in Appendix A [TR 2024].

3.3 A Higher-Order Function with Unresolved Try-Catch Logic Construct

There are situations when try-catch constructs are not sufficiently instantiated for reduction to occur. An example is the higher-order function *foo* with its corresponding *ESL* specification, as shown in Fig. 11. Here, the try-catch construct cannot be *directly* eliminated since its body contains a relation $f(res)$ that is yet to be instantiated. Nevertheless, our verification rules can modularly verify the specification of such functions, due to our adoption of try-catch logic construct.

```
1 let foo f : int (* foo(f,r) = try (∃res·f(res)) catch { Label k → k(5,r) } *)
2 = match f() with
3   | effect Label k -> resume k 5
4
5 let goo () : int (* goo(r) = ∃x·ens[r] x↦0 ∧ r=15 *)
6 = let f = (fun () -> let x = ref 0 in (perform Label) + 10)
7   in foo f
```

Fig. 11. An irreducible try-catch construct in foo and its caller goo

Subsequently, each function call to *foo* may have its unknown argument instantiated (with its summarized specification) which can later facilitate try-catch reduction. An example is function *goo*, which calls *foo* with a lambda argument. As shown in Fig. 12, a specification for *goo* can now be obtained by reducing its instantiated try-catch logic construct. Note that our program verification methodology is *modular* since we only inline summarised specification rather than code, and always perform modular verification on a per-method basis.

$$
\begin{array}{rl}
goo(r) & = \exists f \cdot \ \mathbf{ens}\ f(res) = (\exists x, y \cdot \mathbf{ens}\ x \mapsto 0\,; Label(y); \mathbf{ens}[res]\ res{=}y{+}10)\ ;\ foo(f, r) \\
& \rightsquigarrow \ \mathbf{try}\ \exists res, x, y \cdot \mathbf{ens}\ x \mapsto 0\,; Label(y); \mathbf{ens}[res]\ res{=}y{+}10\ \mathbf{catch}\ \{\, Label\, k \rightarrow k(5, r)\,\} \\
& \rightsquigarrow^* \ \exists x \cdot \mathbf{ens}[r]\ x \mapsto 0 \wedge r{=}15
\end{array}
$$

Fig. 12. Deriving the Specification for Function goo.

3.4 Possibility of Adding Nested Hoare Triple for Function-Type Parameters

Our new *ESL* logic (which will be formally presented in Figure 15) is capable of supporting the full higher-order language. Earlier, we illustrated an example where try-catch reduction *could* get stuck when unknown function parameter call(s) is present. Nevertheless, *ESL* can also support nested Hoare triple for its function-type parameters, if desired, which is a traditional way for fully supporting higher-order functions. For the same *foo* example, a user may instead specify:

$$foo(f, r) \ = \ \mathbf{req}\ \ f(res) = \Phi_1[res]\,; \Phi_2.$$

where "$f(res) = \Phi_1[res]$" is a generalization of the nested Hoare triple which captures an over-approximation of f's behaviors inside *foo*'s precondition. When reasoning with the *goo* method (Fig. 12), we would still have a lambda instantiation, namely:

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

With this new pre-condition, our verifier (see Sec 6) would now need to check a subsumption:

$$\exists\, x, y \cdot \;\; \mathbf{ens}\;\; x \mapsto 0\,;\; \mathit{Label}(y)\,;\; \mathbf{ens}[res]\; res{=}y{+}10 \;\sqsubseteq\; \Phi_1[res].$$

Using such nested Hoare triple for function-type parameter f, try-catch reduction can now occur inside the *foo* method with the help of definition Φ_1. To make this example more concrete, let $\Phi_1[res] = \exists y \cdot \mathcal{N}_1\,;\; \mathit{Label}(y)\,;\; \mathcal{N}_2[res])$, where $\mathcal{N}_1$ and $\mathcal{N}_2$ are some arbitrary normal stages (formally defined in Fig. 16). We can then perform try-catch reduction inside method *foo* as follows:

$$\begin{array}{ll}
& \mathbf{try}\; \exists res \cdot f(res)\; \mathbf{catch}\; \{\mathit{Label}\, k \rightarrow k(5, r)\} \\
=\; & \mathbf{try}\; \exists res \cdot \Phi_1[res]\; \mathbf{catch}\; \{\mathit{Label}\, k \rightarrow k(5, r)\} \\
=\; & \mathbf{try}\; \exists res, y \cdot \mathcal{N}_1\,;\; \mathit{Label}(y)\,;\; \mathcal{N}_2[res]\; \mathbf{catch}\; \{\mathit{Label}\, k \rightarrow k(5, r)\} \\
\rightsquigarrow^*\; & \exists y \cdot \mathcal{N}_1\,;\; \mathbf{ens}\; y{=}5\,;\; \mathcal{N}_2[r].
\end{array}$$

With this elimination of try-catch construct, our new specification for *foo* would be:

$$foo(f, r) \;=\; \exists \mathcal{N}_1, \mathcal{N}_2 \cdot \mathbf{req}\, f(res) = (\exists y \cdot \mathcal{N}_1\,;\; \mathit{Label}(y)\,;\; \mathcal{N}_2[res])\;;\; \exists y \cdot \mathcal{N}_1\,;\; \mathbf{ens}\; y{=}5\,;\; \mathcal{N}_2[r]).$$

However, this new specification for *foo* is actually more verbose and also less precise than the specification we provided in Fig. 11. It is less precise as pre-condition "**req** $f(res) = \Phi_1[res]$" is stronger than "**req** *true*" used implicitly in our original *ESL* specification for *foo*. In both cases, the full higher-order language features are supported by *ESL* but with different degrees of precision.

4 TARGET LANGUAGE AND SPECIFICATIONS

In this section, we define the syntax and semantics of the target language. For the *ESL* specification, we start with a general form, denoted as φ, then present its normalized form, denoted as Φ.

(*Program*)	$\mathcal{P}$	::=	$spec^*\ e$
(*Specifications*)	$spec$	::=	$lemma \mid predicate$
(*Try-Catch Lemma*)	$lemma$	::=	$match[\delta]\ f(x^*, r)\ \#\ \Phi\ with\ \mathcal{H}_\Phi \sqsubseteq \Phi_{inv}$
(*Predicates*)	$predicate$	::=	$g(x^*, r) = \Phi \mid rec\ g(x^*, r) = \Phi$
(*Handler*)	$\mathcal{H}$	::=	$\{x \rightarrow e\} \uplus ops$
(*Operation Cases*)	ops	::=	$\emptyset \mid \{E(x)k \rightarrow e\} \uplus ops$
(*Values*)	v	::=	$c \mid (\lambda\, x^* \rightarrow e) :: \Phi$
(*Expressions*)	e	::=	$v \mid x \mid let\ x{=}e_1\ in\ e_2 \mid if\ x\ then\ e_1\ else\ e_2 \mid f(x^*) \mid x_1{:=}x_2 \mid$ $!x \mid ref(x) \mid assert\ (P) \mid perform\ E(x) \mid match[\delta]\ e\ with\ \mathcal{H}$

(*Constant*) c $\quad x, y, r, f, g, k \in var$ $\quad$ (*Effect Labels*) $E \in \Sigma$ $\quad \delta \in \{s, d\}$ $\quad P{=}\sigma \wedge \pi$

Fig. 13. Syntax of the Target Language.

4.1 Syntax of Target Language

We target an ML-like call-by-value, higher-order core language with primitive mutable state and user-defined algebraic effects and handlers, defined in Fig. 13. A program $\mathcal{P}$ comprises a list of specifications and an expression e, where $*$ superscript denotes a finite, possibly empty list of items. Function definitions are represented using *let* binding and *lambda* expressions.

Specifications are try-catch lemmas or predicate definitions. Try-catch *lemmas* aid inductive proofs for behaviors of handlers, and given lemmas are automatically proved before being applied. Recursively defined predicates, where g occurs in Φ, are explicitly marked with the keyword *rec*. We use $\mathcal{H}_\Phi$ to denote the specification for a handler $\mathcal{H}$. The syntax of the specification formulae Φ is given in Sec. 4.3. Values include constants c (including integers, boolean values, and the unit value ()), and lambda expressions $(\lambda\, x^* \rightarrow e) :: \Phi$, which are closures with annotated/inferred specifications. Expressions consist of values, variables, let bindings, conditionals, function application, heap operations, assertions (P is in separation logic , containing a spatial conjunction of heap σ and pure π formulae, see Figure 15 later) and constructs for performing and handling effects.

The expression *perform* $E(x)$ invokes effect E (e.g., to read a file) with an argument x (e.g., file's location), which is analogous to raising an exception: when executed, evaluation is suspended, and control is transferred to the *nearest enclosing handler* for E. While raising an exception aborts a computation, performing an effect suspends it, passing the handler a continuation k. The handler can use k to *resume* the computation with some result (e.g., the contents of the file), which would be transferred to the suspended computation as the result of the *perform* statement.

The construct $match[\delta]\ e\ with\ \mathcal{H}$ wraps the expression e in an effect handler $\mathcal{H}$. We use δ to distinguish shallow and deep handlers: s for shallow handlers and d for deep handlers. A shallow handler serves its purpose at most once: after it has handled one effect, it disappears. A deep handler is persistent: it remains installed (as the topmost frame of the captured continuation [Hillerström and Lindley 2018; Kammar et al. 2013]) to handle any number of raised effects. Each handler consists of a *normal return clause* $(x \rightarrow e)$, which is used if the expression terminates without any effects, and a set of *operation cases ops* handling different effect labels, in which the variable k provides access to the continuation, as a first-class value. This paper provide supports for both types of handlers.

4.2 Operational Semantics of Core Language

We define the operational semantics using a big-step reduction relation $[S, h, e] \longrightarrow [S_1, h_1, R^r]$ in Fig. 14, denoting from an initial *store* S and *heap* h, e reduces to some runtime *outcome* R^r, changing the store and heap to S_1, h_1. Each store S is a partial map $var \rightharpoonup val$, where *var* is the set of (immutable) program variables and *val* is the set of *primitive values* – the set of values that can occur syntactically, augmented with *memory locations* ℓ and *closures* $(\lambda x \rightarrow e, S)$, which are a pair of a lambda expression and a store that gives values to its (immutable) free variables. The heap h is a partial map $loc \rightharpoonup val$. Evaluation results R^r take one of the forms given at the top of Fig. 14 – they are either a *normal return* of a primitive value $Norm(v)$, an occurrence of an *unhandled effect* $Eff(E(v), e_k)$ with argument v and *continuation* e_k (a lambda/closure), or an error *Err*, which occurs on assertion failure. The inclusion of unhandled effects as an evaluation outcome allows handlers and continuations to be directly expressible in our semantics.

Variables are read from the store, while constants evaluate to themselves. Lambda expressions evaluate to closures, capturing the current store. There are four cases for *let*, which serves to support recursion and sequence evaluation, depending on whether evaluation of e_1 produces a value, an error, or an unhandled effect; evaluation either continues, terminates, or suspends with a continuation. Conditionals are standard, and application of a closure restores its captured store before evaluating it. Next are heap operations and assertion, followed by rules for handling effects.

$$R^r \;::=\; Norm(v) \mid Eff(E(v), e_k) \mid Err$$

$$[S,h,x] \longrightarrow [S,h,Norm(S(x))] \quad (OP\text{-}Var)$$

$$[S,h,c] \longrightarrow [S,h,Norm(c)] \quad (OP\text{-}Constant)$$

$$[S,h,\lambda\, y^* \to e] \longrightarrow [S,h,Norm((\lambda y^* \to e, S))] \quad (OP\text{-}Lambda)$$

$$[S,h,let\ x{=}E\ in\ e_2] \longrightarrow [S_3,h_3,R^r]\ \boxed{\text{if}}\ [S_1,h,e_2] \longrightarrow [S_3,h_3,R^r]$$
$$\text{where } E = \lambda\, y^* \to e_1 \quad S_1 = S{+}[x{:=}(E,S_1)] \quad (OP\text{-}Let\text{-}Rec)$$

$$[S,h,let\ x{=}e_1\ in\ e_2] \longrightarrow [S_2,h_2,R^r]\ \boxed{\text{if}}\ [S,h,e_1] \longrightarrow [S_1,h_1,Norm(v)] \text{ and}$$
$$[S{+}[x{:=}v],h_1,e_2] \longrightarrow [S_2,h_2,R^r] \quad (OP\text{-}Let\text{-}Norm)$$

$$[S,h,let\ x{=}e_1\ in\ e_2] \longrightarrow [S_1,h_1,Err]\ \boxed{\text{if}}\ [S,h,e_1] \longrightarrow [S_1,h_1,Err] \quad (OP\text{-}Let\text{-}Err)$$

$$[S,h,let\ x{=}e_1\ in\ e_2] \longrightarrow [S,h_1,Eff(E(v),(\lambda\, y \to let\ x{=}e_k(y)\ in\ e_2,S))]$$
$$\boxed{\text{if}}\ [S,h,e_1] \longrightarrow [S_1,h_1,Eff(E(v),e_k)] \quad (OP\text{-}Let\text{-}Eff)$$

$$[S,h,if\ x\ then\ e_1\ else\ e_2] \longrightarrow [S_1,h_1,R^r]\ \boxed{\text{if}}\ S(x){=}true \text{ and } [S,h,e_1] \longrightarrow [S_1,h_1,R^r] \quad (OP\text{-}If\text{-}true)$$

$$[S,h,if\ x\ then\ e_1\ else\ e_2] \longrightarrow [S_1,h_1,R^r]\ \boxed{\text{if}}\ S(x){=}false \text{ and } [S,h,e_2] \longrightarrow [S_1,h_1,R^r] \quad (OP\text{-}If\text{-}false)$$

$$[S,h,f(x^*)] \longrightarrow [S_1,h_1,R^r]\ \boxed{\text{if}}\ S(f) = (\lambda y^* \to e, S_\lambda) \text{ and}$$
$$[S{+}S_\lambda,h,e[(x/y)^*]] \longrightarrow [S_1,h_1,R^r] \quad (OP\text{-}Apply)$$

$$[S,h,x_1{:=}x_2] \longrightarrow [S,h[S(x_1){:=}S(x_2)],Norm(())]\ \boxed{\text{if}}\ S(x_1) \in dom(h) \quad (OP\text{-}Assign)$$

$$[S,h,!x] \longrightarrow [S,h,Norm(h(S(x)))]\ \boxed{\text{if}}\ S(x) \in dom(h) \quad (OP\text{-}Deref)$$

$$[S,h,ref(x)] \longrightarrow [S,h{+}[\ell{:=}S(x)],Norm(\ell)]\ \boxed{\text{if}}\ \ell \notin dom(h) \quad (OP\text{-}Ref)$$

$$[S,h,assert\ \sigma{\wedge}\pi] \longrightarrow [S,h,Norm(())]\ \boxed{\text{if}}\ \exists h_1 \cdot h_1 {\subseteq} h \text{ and } S,h_1 \models \sigma{\wedge}\pi \quad (OP\text{-}Assert)$$

$$[S,h,assert\ \sigma{\wedge}\pi] \longrightarrow [S,h,Err]\ \boxed{\text{if}}\ \forall h_1 \cdot h_1 {\subseteq} h \Rightarrow S,h_1 \not\models \sigma{\wedge}\pi \quad (OP\text{-}Assert\text{-}Err)$$

$$[S,h,match[\delta]\ e\ with\ \mathcal{H}] \longrightarrow [S_2,h_2,R^r]\ \boxed{\text{if}}\ [S,h,e] \longrightarrow [S_1,h_1,Norm(v)] \text{ and}$$
$$(x \to e_n) \in \mathcal{H} \text{ and } [S_1,h_1,e_n[v/x]] \longrightarrow [S_2,h_2,R^r] \quad (OP\text{-}Ret)$$

$$[S,h,match[s]\ e\ with\ \mathcal{H}] \longrightarrow [S_2,h_2,R^r]\ \boxed{\text{if}}\ [S,h,e] \longrightarrow [S_1,h_1,Eff(E(x),e_k)] \text{ and}$$
$$(E(x)k \to e_h) \in \mathcal{H} \text{ and } (x_1 \to e_n) \in \mathcal{H} \text{ and}$$
$$[S_1{+}[k{:=}(\lambda\, y \to let\ x_1{=}e_k(y)\ in\ e_n)],h_1,e_h[v/x]]$$
$$\longrightarrow [S_2,h_2,R^r] \quad (OP\text{-}Shallow)$$

$$[S,h,match[d]\ e\ with\ \mathcal{H}] \longrightarrow [S_2,h_2,R^r]\ \boxed{\text{if}}\ [S,h,e] \longrightarrow [S_1,h_1,Eff(E(v),e_k)] \text{ and}$$
$$S_k = S_1{+}[k{:=}(\lambda\, y \to match[d]\ e_k(y)\ with\ \mathcal{H},S)] \text{ and}$$
$$(E(x)k \to e_h) \in \mathcal{H} \text{ and}$$
$$[S_k,h_1,e_h[v/x]] \longrightarrow [S_2,h_2,R^r] \quad (OP\text{-}Deep)$$

$$[S,h,match[\delta]\ e\ with\ \mathcal{H}] \longrightarrow [S,h_1,Eff(E(v),\lambda\, y \to match[\delta]\ e_k(y)\ with\ \mathcal{H})]$$
$$\boxed{\text{if}}\ [S,h,e] \longrightarrow [S_1,h_1,Eff(E(v),e_k)] \text{ and}$$
$$E \notin dom(\mathcal{H}) \quad (OP\text{-}Unhandled)$$

$$[S,h,match[\delta]\ e\ with\ \mathcal{H}] \longrightarrow [S_1,h_1,Err]\ \boxed{\text{if}}\ [S,h,e] \longrightarrow [S_1,h_1,Err] \quad (OP\text{-}Match\text{-}Err)$$

$$[S,h,perform\ E(x)] \longrightarrow [S,h,Eff(E(S(x)),(\lambda\, y \to y,[]))] \quad (OP\text{-}Perform)$$

Fig. 14. Big-Step Operational Semantics for Core Language with Algebraic Effects.

In (*OP-Ret*), if the evaluation of the scrutinee e produces a value, the return clause of the handler is executed. If it produces an effect, there are two cases depending on whether the handler is deep or shallow. In (*OP-Shallow*), execution continues in the body of the appropriate handler case e_h, with two arguments bound: the argument v given when the effect was *perform*ed, and the continuation k carried by the effect. The rule for deep handlers (*OP-Deep*) differs in one crucial way: the continuation k is wrapped with an identical handler, so subsequent effects from the continuation will be handled under the same handler. The next two rules cover handled effects and scrutinees which terminate with errors. Finally, (*OP-Perform*) is where unhandled effects originate, with an identity continuation that is successively extended by let-bindings.

4.3 Syntax of Specification Language

We first define the most general form of *ESL*, using φ, in Fig. 4.3.1 to facilitate a simpler semantics model; then define a normalized form of *ESL*, using Φ, in Fig. 4.3.2, to facilitate more straightforward forward reasoning, try-catch reduction, and the entailment checking. The soundness of our verifier builds on the fact that every normalized *ESL* formula has a correspondence in the general format.

4.3.1 General ESL. The syntax of φ is shown in Fig. 15. The first two constructs are familiar pre/post specifications, which compactly represent program states. In particular, the **ens**ure construct contains a state Q and explicitly indicates the return variable r. Those **req**uire and **ens**ure constructs can be composed using sequential composition and disjunction to represent sets of program traces. Existential variables may be used to capture intermediate values which arise along such traces.

Finally, imperative, effectful, and higher-order behavior that is difficult to summarize using pre/post specifications can be modeled by the following three new constructs. They give rise to the idea of *stages*, as they stratify traces which can otherwise be compacted into pre/post specifications.

- *Effect constructs* like $E(x, r)$, describe occurrences of unhandled algebraic effects, with arguments x^* and a resumed variable r.
- *Predicate constructs* like $g(x^*, r)$, describe calls to higher-order function parameters g, whose (algebraic or imperative) effects are, at the point, unknown.
- *Try-catch constructs* like $\mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$, describe the state resulting from handling effects occurring in some formula φ under a handler whose cases are abstractly specified as *ESL* formulae.

$$
\begin{array}{rrcl}
(\mathit{ESL}) & \varphi & ::= & \mathbf{req}\ P \mid \mathbf{ens}[r]\ Q \mid \varphi;\varphi \mid \varphi \vee \varphi \mid \exists\, x^* \cdot \varphi \mid \\
 & & & E(x, r) \mid f(x^*, r) \mid \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi \\
(\mathit{Handle\ Spec.}) & \mathcal{H}_\Phi & ::= & \{pat_i \rightarrow \varphi_i\}_{i=1}^{n} \\
(\mathit{Pattern}) & pat & ::= & x \mid E(x)\ k \\
(\mathit{State}) & P, Q & ::= & \exists x^* \cdot \sigma \wedge \pi \\
(\mathit{Heap}) & \sigma & ::= & emp \mid x \mapsto v \mid \sigma_1 * \sigma_2 \\
(\mathit{Terms}) & t & ::= & v \mid t_1{+}t_2 \mid t_1{-}t_2 \\
(\mathit{Pure}) & \pi & ::= & true \mid false \mid bop(t_1, t_2) \mid f(x^*, r){=}\varphi \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \exists x \cdot \pi
\end{array}
$$

Fig. 15. Syntax of General *ESL* (Effectful Specification Logic).

We describe program states using separation logic formulae σ from the symbolic heap fragment, which can be: a predicate *emp*, which models an empty heap; singleton heap predicates $x \mapsto v$, which describes a location in memory pointed to v by x; formula $x \mapsto \_$, which means that x is allocated; separating conjunction $\sigma_1 * \sigma_2$, and existential quantification over values (including locations). We use π to denote pure formulae, which capture arithmetic and boolean constraints on

program variables, as well as definitions of staged specifications for functions. Binary relations $bop(t_1, t_2)$ include $=, >, <, \geq$ and $\leq$. Terms are values or additions/subtractions of terms.

$$
\begin{array}{rrcl}
(\mathit{ESL}) & \Phi & ::= & \theta \mid \Phi \vee \Phi \\
(\mathit{Staged\ Flow}) & \theta & ::= & \mathcal{E}^* \,;\, \mathcal{N} \\
(\mathit{Effect/Pred./TryCatch\ Stages}) & \mathcal{E} & ::= & \mathcal{N} \,;\, \mathcal{O} \\
(\mathit{Uninterpreted\ Predicates}) & \mathcal{O} & ::= & E(x, r) \mid g(x^*, r) \mid \mathbf{try}[\delta](\Phi)\ \mathbf{catch}\ \mathcal{H}_\Phi \\
(\mathit{Normal\ Stage}) & \mathcal{N} & ::= & \exists x^* \cdot \mathbf{req}\ P\ \mathbf{ens}\ Q
\end{array}
$$

Fig. 16. Syntax of Normalized *ESL*.

4.3.2 Normalized ESL. In most of this paper, we focus on *ESL* formulae occurring in a more structured form Φ, shown in Fig. 16. The rules of Sec. 5.1 operate on and maintain this form, and additional normalization steps for it are covered in Sec. 5.2. The intuition behind this form is that we would like to reason about straight-line program paths described using *staged flows*, which are sequences of stages (and thus describe *segments* in traces). Top-level *ESL* formulae Φ are disjunctions of flows θ. Staged flows are in the form of $(\mathcal{E}^* \,;\, \mathcal{N})$, which contains a prefix of stages $\mathcal{E}^*$ followed by a final normal stage $\mathcal{N}$. Each $\mathcal{E}$ stage contains a normal stage followed by an uninterpreted predicate, $\mathcal{O}$, which indicates unhandled effects, higher-order calls that are not yet instantiated or irreducible try-catch constructs, respectively. For example, an *effect stages* like $\mathcal{N} \,;\, E(x, r)$, contains a normal stage $\mathcal{N}$, which describes the state just before the occurrence of an unhandled effect.

Discussion. Traditional separation logic specifications denoted by (**req** P **ens** Q) can be captured by a single normal stage. Therefore, *ESL* formulae describe program traces in a compact form, revealing only *interesting* points along them. This design enables careful specification of effectful and imperative program behaviors. In the next section, we formalize the semantics of *ESL* formulae.

4.4 Semantic Model of Stages

We assume a standard separation logic *models* relation $S, h \models \sigma \wedge \pi$, which holds iff the state S, h satisfies the heap formula $\sigma \wedge \pi$. Other standard notation for heaps is used: $dom(h)$ is the domain of heap h, $h_1 \circ h_2 = h$ is the disjoint union, i.e., given $dom(h_1) \cap dom(h_2) = \emptyset$, $h_1 \cup h_2 = h$, $S + [x := v]$ and $S \backslash \{x\}$ respectively denote store extension and removal of variables. $S_1 + S_2$ denotes a store merge where bindings in S_2 take precedence. The same operations apply to heaps, which may additionally be updated, denoted by $h[x := v]$. We write $h_1 \subseteq h_2$ to denote that h_1 is a subheap of h_2, i.e., $\exists h_3 \cdot h_1 \circ h_3 = h_2$.

The semantics of a staged formula φ is given as a *models* relation $[C, S, h] \rightsquigarrow_m [C_1, S_1, h_1, R^c] \models \varphi$. Since staged formulae describe execution traces, the *models* relations holds iff in the starting state S, h, if the program described abstractly by φ terminates, it does so in a final state S_1, h_1 with compile-time (set-based) outcome R^c, which is either of the form $Norm(r)$, indicating the normal return of a value via variable r, $Eff(E(v, r), \varphi_1)$, indicating an unhandled effect E with argument v, return variable r and continuation described by φ_1, or $\top$, indicating an indeterminate result (that includes *Err* error). C (resp. C_1) is a boolean value that is *false* (✗) iff a precondition failure has *possibly* occurred prior to (resp. after) the execution of φ, otherwise it is *true* (✓). If a precondition failure occurs during the execution of φ, the result in the final state becomes indeterminate ($\top$), and further execution vacuously succeeds, as shown in the first rule in Fig. 17.

The meaning of **req** $\sigma \wedge \pi$ is given by the next two rules: it requires a heaplet h_2 satisfying $\sigma \wedge \pi$ to be part of the initial heap h and removes it, analogous to a standard separation logic precondition.

$[\boldsymbol{\times}, S, h] \rightsquigarrow_m [\boldsymbol{\times}, S, h, \top] \models \varphi$

$R^c ::= Norm(r) \mid Eff(E(r), \varphi) \mid \top$

$[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S, h_1, Norm(\_)] \models \mathbf{req}\ \sigma \wedge \pi$ *iff* $h_1 \subseteq h$ and $S, h_1 \models \sigma \wedge \pi$

$[\checkmark, S, h] \rightsquigarrow_m [\boldsymbol{\times}, S, h, \top] \models \mathbf{req}\ \sigma \wedge \pi$ *iff* $\forall h_1 \cdot h_1 \subseteq h \Rightarrow S, h_1 \not\models \sigma \wedge \pi$

$[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S, h \circ h_2, Norm(r)] \models \mathbf{ens}[r]\ \sigma \wedge \pi$ *iff* $dom(h) \cap dom(h_2) = \emptyset$ and $S, h_2 \models \sigma \wedge \pi$

$[\checkmark, S, h] \rightsquigarrow_m [C, S_2, h_2, R^c] \models \varphi_1; \varphi_2$ *iff* $\exists S_1, h_1 \cdot [\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_1, h_1, Norm(r)] \models \varphi_1$ and $[\checkmark, S_1, h_1] \rightsquigarrow_m [C, S_2, h_2, R^c] \models \varphi_2$

$[\checkmark, S, h] \rightsquigarrow_m [\boldsymbol{\times}, S_1, h_1, \top] \models \varphi_1; \varphi_2$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [\boldsymbol{\times}, S_1, h_1, \top] \models \varphi_1$

$[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_1, h_1, R^c] \models \varphi_1; \varphi_2$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_1, h_1, Eff(E(x^*, r), \varphi_0)] \models \varphi_1$ and $R^c = Eff(E(x^*, r), (\varphi_0; \varphi_2))$

$[\checkmark, S, h] \rightsquigarrow_m [C_1 \wedge C_2, S_3, h_3, R_3] \models \varphi_1 \vee \varphi_2$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [C_1, S_1, h_1, R_1] \models \varphi_1$ *and* $[\checkmark, S, h] \rightsquigarrow_m [C_2, S_2, h_2, R_2] \models \varphi_2$ *and* $(S_3, h_3, R_3) \in \{(S_1, h_1, R_1), (S_2, h_2, R_2)\}$

$[\checkmark, S, h] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \exists x \cdot \varphi$ *iff* $\exists\ v \cdot [\checkmark, S{+}[x{:=}v], h] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \varphi$

$[\checkmark, S, h] \rightsquigarrow_m [C, S_1, h_1, R^c] \models f(v^*, r')$ *iff* $(f(x^*, r) = \varphi) \in S$ *and* $[\checkmark, S, h] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \varphi[v^*/x^*, r'/r]$

$[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S, h, R^c] \models E(x, r)$ *iff* $R^c = Eff(E(x, r), \mathbf{ens}[r]\ emp)$

$[\checkmark, S, h] \rightsquigarrow_m [\boldsymbol{\times}, S_1, h_1, \top] \models \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [\boldsymbol{\times}, S_1, h_1, \top] \models \varphi$

$[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_1, h_1, R^c] \models \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_1, h_1, Eff(E(x, r), \varphi_1)] \models \varphi$ $E \notin dom(\mathcal{H}_\Phi)$ and $R^c = Eff(E(x, r), \mathbf{try}[\delta](\varphi_1)\ \mathbf{catch}\ \mathcal{H}_\Phi)$

$[\checkmark, S, h] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \mathbf{try}[\delta](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_2, h_2, Norm(r)] \models \varphi$ *and* $(x \rightarrow \varphi_n) \in \mathcal{H}_\Phi$ *and* $[\checkmark, S_2, h_2] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \varphi_n[r/x]$

$[\checkmark, S, h] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \mathbf{try}[s](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_2, h_2, Eff(E(x, r), \varphi_1[r_n])] \models \varphi$ $(E(y)k \rightarrow \varphi_2) \in \mathcal{H}_\Phi$ and $(x_1 \rightarrow \varphi_n) \in \mathcal{H}_\Phi$ and $S_k = S_2{+}[k{:=}\lambda\ (r, r_c) \rightarrow \varphi_1[r_n]; (\varphi_n[r_n/x_1])[r_c]]$ and $[\checkmark, S_k, h_2] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \varphi_2[x/y]$

$[\checkmark, S, h] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \mathbf{try}[d](\varphi)\ \mathbf{catch}\ \mathcal{H}_\Phi$ *iff* $[\checkmark, S, h] \rightsquigarrow_m [\checkmark, S_2, h_2, Eff(E(x, r), \varphi_1)] \models \varphi$ $(E(y)k \rightarrow \varphi_2) \in \mathcal{H}_\Phi$ and $S_k = S_2{+}[k{:=}\lambda\ (r, r_c) \rightarrow (\mathbf{try}[d](\varphi_1)\ \mathbf{catch}\ \mathcal{H}_\Phi)[r_c]]$ and $[\checkmark, S_k, h_2] \rightsquigarrow_m [C, S_1, h_1, R^c] \models \varphi_2[x/y]$

Fig. 17. Semantics of Staged Formulae with Effects and Try-Catch Handlers.

If there is no such heaplet, a precondition failure occurs. **ens** $\sigma \wedge \pi$ plays a dual role, creating a heaplet h_2. Like the cases for *let* in the operational semantics, there are three cases for sequencing depending on whether the first formula results in a value, effect, or error, with errors propagating via precondition failures. Disjunction chooses between one of two flows and ensures that precondition failure *cannot occur* if *both* C_1 and C_2 have *no* precondition failures. Existentials add variables to the store with existential values; "reading" of these variables from the store occurs via the separation logic *models* relation. The semantics of predicate stages is simply that of the function they stand for. As we define the operational semantics, it is reasonable to expect that to execute programs, they must be *closed*, and predicate stages refer to functions defined in the program. The remaining cases handle effect stages. Like the case for *perform* in the operational semantics, an effect stage occurring on its own is unhandled, resulting in an unhandled effect with an identity continuation, represented as a trivial staged formula. The rules for *try* cover the cases where the scrutinee staged formula fails or produces an unhandled effect, a result, or a handled effect under a shallow or deep handler, in order. They are largely similar to the operational semantics, as the *try-catch* construct is a direct symbolic analogue of the *match-with* statement.

5 FORWARD VERIFICATION

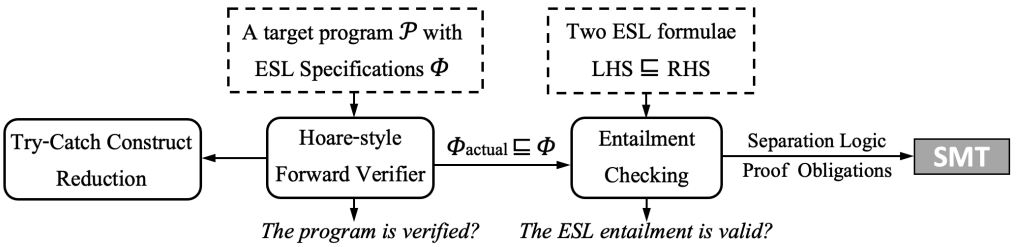

Fig. 18. System Overview

Fig. 18 presents an overview of our automated verification system. Our main technical contributions are captured in the rounded boxes: a Hoare-style forward verifier, a reduction for try-catch constructs and an entailment checker. The input of the forward verifier is a target program $\mathcal{P}$, and its functions are annotated with the *ESL* specifications Φ. The input of the entailment checking is a pair of *ESL* formulae: LHS and RHS, referring to the entailment LHS $\sqsubseteq$ RHS to be checked *(LHS and RHS refer to left/right-hand-side effects respectively)*. The *ESL* entailment relation $\sqsubseteq$ is formally defined in Sec. 6. The workflow of our automated verification system is as follows:

(1) The forward verifier takes a program $\mathcal{P}$, which contains a set of functions annotated with specifications Φ, and predefined lemmas. For each function, a modular verification – where functions can be replaced by their already-verified specifications – computes the actual behaviors of the function body, denoted by Φ_{actual}, using a set of forward rules, defined in Sec. 5.1.
(2) The forward verifier employs a set of reduction and normalization rules, defined in Sec. 5.2, to eliminate try-catch logic constructs when possible.
(3) Taking Φ_{actual} and Φ, the back-end checker proves/disproves the entailment $\Phi_{actual} \sqsubseteq \Phi$. We establish a set of entailment rules in Sec. 6. Separation logic proof obligations are reduced [Chin et al. 2011; Piskac et al. 2013] to decidable first-order theory that fits well into the satisfiability modulo theories (SMT) framework. We explain the entailment checking in Sec. 6.

Next, we elaborate on the forward reasoning rules in Sec. 5.1, the try-catch reduction in Sec. 5.2, and present the soundness proofs in Sec. 5.3.

5.1 Forward Verification Rules

We formalize a set of syntax-directed forward rules for the target language. The forward reasoning is in the form of Hoare-style triples: $\vdash \{\mathcal{N}\}\ e\ \{\Phi\}$. Under a partial correctness interpretation, which we adopt in this paper, the triple means that if $\mathcal{N}$ describes the latest normal stage before executing e, if e terminates, Φ describes the staged flows that will be triggered after. The verification is initialized with a fresh normal stage, i.e., **ens** *emp*, as a short-hand for (**req** *emp* **ens**[_] *emp*).

$$\frac{\{\mathcal{N}\}\ e\ \{\Phi\}}{\{\mathcal{E}^*\,;\,\mathcal{N}\}\ e\ \{\mathcal{E}^*\,;\,\Phi\}}\ [\textit{FV-Frame}] \qquad \frac{\{x^*\}\cap fvars(e)=\{\} \quad \{\mathbf{req}\ P\ \mathbf{ens}\ Q\}\ e\ \{\Phi\}}{\{\exists x^*\cdot \mathbf{req}\ P\ \mathbf{ens}\ Q\}\ e\ \{\exists x^*\cdot\Phi\}}\ [\textit{FV-Ex}]$$

$$\frac{\{\Phi_1\}\ e\ \{\Phi_3\} \qquad \{\Phi_2\}\ e\ \{\Phi_3\}}{\{\Phi_1\vee\Phi_2\}\ e\ \{\Phi_3\}}\ [\textit{FV-Disj}]$$

$$\frac{\mathcal{N}=\mathbf{req}\ P\ \mathbf{ens}[\_]\ Q \qquad v ::= c \mid x}{\{\mathcal{N}\}\ v\ \{\exists r\cdot \mathbf{req}\ P\ \mathbf{ens}[r]\ Q\wedge r{=}v\}}\ [\textit{FV-Var-Const}]$$

$$\frac{\{\mathcal{N}\}\ e_1\ \{\Phi_1[r]\} \qquad \{\Phi_1\wedge x{=}r\}\ e_2\ \{\Phi_2\}}{\{\mathcal{N}\}\ \textit{let}\ x{=}e_1\ \textit{in}\ e_2\ \{\exists x\cdot\Phi_2\}}\ [\textit{FV-Let}]$$

$$\frac{\{\mathcal{N}\}\ \textit{perform}\ E(x) \quad \{\exists r\cdot\mathcal{N}\,;\,E(x,r)\}}{\{\mathcal{N}\}\ (\lambda\, x^*{\to}e)::\Phi[r]\ \{\exists f\cdot \mathbf{req}\ P\ \mathbf{ens}[f]\ Q\wedge f{=}(\lambda\,(x^*,r){\to}\Phi[r])\}}\ [\textit{FV-Perform}]$$

$$\frac{\mathcal{N}{=}\mathbf{req}\ P\ \mathbf{ens}\ Q \qquad \{\mathbf{ens}\ \textit{Pure}(Q)\}\ e\ \{\exists r'\cdot\Phi'[r']\} \qquad \Phi'[r']\sqsubseteq\Phi[r'/r]}{\{\mathcal{N}\}\ (\lambda\, x^*{\to}e)::\Phi[r]\ \{\exists f\cdot \mathbf{req}\ P\ \mathbf{ens}[f]\ Q\wedge f{=}(\lambda\,(x^*,r){\to}\Phi[r])\}}\ [\textit{FV-Lambda-Def}]$$

$$\frac{\{\mathcal{N}\wedge x\}\ e_1\ \{\Phi_1\} \qquad \{\mathcal{N}\wedge\neg x\}\ e_2\ \{\Phi_2\}}{\{\mathcal{N}\}\ \textit{if}\ x\ \textit{then}\ e_1\ \textit{else}\ e_2\ \{\Phi_1\vee\Phi_2\}}\ [\textit{FV-If-Else}]$$

$$\frac{\mathcal{N}=\mathbf{req}\ P\ \mathbf{ens}\ Q}{\{\mathcal{N}\}\ \textit{ref}(x)\ \{\exists y\cdot \mathbf{req}\ P\ \mathbf{ens}[y]\ Q*(y\mapsto x)\}}\ [\textit{FV-Ref}]$$

$$\frac{(f(x^*,r)=\Phi)\in\textit{Pure}(\mathcal{N})}{\{\mathcal{N}\}\ f(y^*)\ \{\exists r\cdot\mathcal{N}\,;\,\Phi[y^*/x^*]\}}\ [\textit{FV-Pred}]$$

$$\frac{\mathcal{N}=\mathbf{req}\ P\ \mathbf{ens}\ Q \qquad Q*?P_A\vdash_{\textit{Bi-ab}}(\sigma\wedge\pi)*?\_}{\{\mathcal{N}\}\ \textit{assert}\ (\sigma\wedge\pi)\ \{\exists r\cdot \mathbf{req}\ P*P_A\ \mathbf{ens}[r]\ Q*P_A\wedge r{=}()\}}\ [\textit{FV-Assert}]$$

$$\frac{(\textit{rec}\ f(x^*,r)=\Phi)\in\textit{Pure}(\mathcal{N})}{\{\mathcal{N}\}\ f(y^*)\ \{\exists r\cdot\mathcal{N}\,;\,f(y^*,r)\}}\ [\textit{FV-Rec-Pred}]$$

$$\frac{\mathcal{N}=\mathbf{req}\ P\ \mathbf{ens}\ Q \qquad Q*?P_A\vdash_{\textit{Bi-ab}}(x\mapsto z)*?Q_F}{\{\mathcal{N}\}\ !x\ \{\exists z\cdot \mathbf{req}\ P*P_A\ \mathbf{ens}[z]\ Q_F*(x\mapsto z)\}}\ [\textit{FV-Read}]$$

$$\frac{(f(x^*,r)=\ldots)\notin\textit{Pure}(\mathcal{N})}{\{\mathcal{N}\}\ f(y^*)\ \{\exists r\cdot\mathcal{N}\,;\,f(y^*,r)\}}\ [\textit{FV-Call-Unknown}]$$

$$\frac{\mathcal{N}=\mathbf{req}\ P\ \mathbf{ens}\ Q \qquad Q*?P_A\vdash_{\textit{Bi-ab}}(x{\mapsto}\_)*?Q_F}{\{\mathcal{N}\}\ x:=y\ \{\exists r\cdot \mathbf{req}\ P*P_A\ \mathbf{ens}[r]\ Q_F*(x{\mapsto}y)\wedge r{=}()\}}\ [\textit{FV-Write}]$$

$$\frac{\{\mathcal{N}\}\ e\ \{\Phi\} \qquad \forall i{\in}\{1..n\}\{\mathbf{ens}\ \textit{Pure}(\Phi)\}\ e_i\ \{\Phi_i\} \qquad \mathcal{H}_\Phi=\{pat_i\to\Phi_i\}_{i=1}^n \qquad \mathbf{try}[\delta](\Phi)\ \mathbf{catch}\ \mathcal{H}_\Phi\rightsquigarrow\Phi'\ \ \text{(cf. Sec. 5.2)}}{\{\mathcal{N}\}\ \textit{match}[\delta]\ e\ \textit{with}\ \{pat_i\Rightarrow e_i\}_{i=1}^n\ \{\Phi'\}}\ [\textit{FV-Match}]$$

Fig. 19. Hoare-style Forward Reasoning Rules

We present key forward Hoare-style reasoning rules in Fig. 19. Three structural rules are first shown to handle the frames on the history flows ([*FV-Frame*]), existential variables ([*FV-Ex*]), and the disjunctions on the starting program state ([*FV-Disj*]).

Rule [*FV-Var-Const*] tracks the result r of latest program state by binding it to a value v, which can be either a variable or a constant. Rule [*FV-Let*] firstly reasons about e_1 and generates an existential variable x, then binds x to the result value of e_1 in its last normal stage and continues to compute the staged flows of the rest of the code. The final state of such sequencing is to concatenate the history flows of e_1 and the flows generated from e_2.

Rule [*FV-Perform*] models each effect as a predicate with an existential result r. As shown in the rule [*FV-Lambda-Def*], given any lambda definition with a specification $\Phi[r]$, modular verification starts by computing the actual behavior of the function body, denoted by $\exists r' \cdot \Phi'[r']$, and check whether it entails its specifications. If the entailment succeeds, the rule binds its specification to a freshly created existential name f. As lambda function may be applied anytime and anywhere, we can only make use of pure information that is available when it was first constructed. Hence, $Pure(Q)$ extracts pure formula that can be used as an assumption for use by the lambda's body. Rule [*FV-If-Else*] computes the staged flows from both branches by extending the state with v bound to *true* and *false*, respectively; then, it disjunctively unions the results. Here, we write $\mathcal{N} \wedge \pi$ to mean $(\exists x^* \cdot \mathbf{req}\ P\ \mathbf{ens}\ Q \wedge \pi)$, if $(\mathcal{N} = \exists x^* \cdot \mathbf{req}\ P\ \mathbf{ens}\ Q)$. Rule [*FV-Pred*] concatenates the instantiated specification for the callee function f to the current state. In cases where the predicate is recursively defined, the rule [*FV-Rec-Pred*] instantiates the predicate without unfolding the definition. In cases where f is unknown in the current program, the rule [*FV-Call-Unknown*] extends the program state with an uninterpreted predicate.

[*FV-Match*] computes the staged flows of e, denoted as Φ; integrates the handler's specification, denoted as $\mathcal{H}_\Phi$; then it employs reduction rules to eliminate the try-catch construct. We elaborate on the reduction for handlers in Sec. 5.2. Here, $Pure(\Phi)$ is meant to propagate the pure information accumulated from the staged flow Φ.

[*FV-Assert*], [*FV-Write*], [*FV-Read*], and [*FV-Ref*] make use of a bi-abduction operator $\vdash_{Bi-ab}$, and propagate the *anti-frame* to the precondition and update the postcondition based on the *frame*.

Bi-abduction. Bi-abduction is a form of logical inference for separation logic to support automated heap-based local reasoning. Usually, an entailment for separation logic like $P \vdash_{sl} Q$ means that P implies Q. A challenge is for the theorem prover to discover a pair of *frame* and *anti-frame* formulae that make the entailment valid. The inference of the frame Q_F and anti-frame P_A in a bi-abduction relation $(P * ?P_A \vdash_{Bi-ab} Q * ?Q_F)$ is now well-understood [Calcagno et al. 2009; Le et al. 2014].

An example of the forward reasoning. Fig. 20 sketches the steps of verifying the *callee* function (defined in Fig. 1). Program behavior is captured in the form of $\{\ \Phi\ \}$. We label the steps from (1) to (7) and mark the applied forward rules in [*gray*].

The initial state in step (1) and the entailment checking in step (7) are obtained by the rule [*FV-Fun*]. The state in step (4) is obtained by applying the rules [*FV-Read*] and [*FV-Write*] in sequence. At step (6), we obtained the actual behaviors Φ_{actual} of *callee*, which can be normalized into one effect-flow followed by one normal-flow, namely $\Phi_{actual}(r') = \mathcal{E}\ ;\ \mathcal{N}$ where,

$$\mathcal{E} = \exists x, ret \cdot \mathbf{ens}\ x{\mapsto}0\ ;\ Label(ret) \quad and \quad \mathcal{N} = \exists z \cdot \mathbf{req}\ x{\mapsto}z \wedge z{+}1{=}1\ \mathbf{ens}[r']\ x{\mapsto}z{+}1 \wedge r'{=}ret{+}2.$$

Deployed normalisation rules. Fig. 21 presents the normalization rules that are used during the reasoning and keep the program states always in a normalized form.

(1) `let callee (): int =` [*FV-Fun*]
$\{\,\mathbf{ens}\ emp\ \}$
(2) `let x = ref 0 in` [*FV-Ref*] [*FV-Let*]
$\{\,\exists x \cdot \mathbf{ens}[x]\ x{\mapsto}0\,\}$
(3) `let ret = perform Label in` [*FV-Perform*]
$\{\,\exists x, ret \cdot \mathbf{ens}\ x{\mapsto}0\ ;\ Label(ret)\,\}$
(4) `x := !x + 1;` [*FV-Read*] [*FV-Write*]
$\{\,\exists x, ret \cdot \mathbf{ens}\ x{\mapsto}0\ ;\ Label(ret);\ \exists z, r \cdot \mathbf{req}\ x{\mapsto}z\ \mathbf{ens}[r]\ x{\mapsto}z{+}1 \wedge r{=}()\,\}$
(5) `assert (!x = 1);` [*FV-Assert*]
$\{\,\exists x, ret \cdot \mathbf{ens}\ x{\mapsto}0\ ;\ Label(ret);\ \exists z, r \cdot \mathbf{req}\ x{\mapsto}z \wedge z{+}1{=}1\ \mathbf{ens}[r]\ x{\mapsto}z{+}1 \wedge r{=}()\,\}$
(6) `ret+2` [*FV-Var*]
$\Phi_{actual}(r') = \exists x, ret \cdot \mathbf{ens}\ x{\mapsto}0\ ;\ Label(ret);\ \exists z \cdot \mathbf{req}\ x{\mapsto}z \wedge z{+}1{=}1\ \mathbf{ens}[r']\ x{\mapsto}z{+}1 \wedge r'{=}ret{+}2$
(7) $\Phi_{actual}(r') \sqsubseteq callee(r_c)[r'/r_c]$ [*FV-Fun*] where $callee(r_c)$ is formally defined in Fig. 2.

Fig. 20. Demonstrating the Forward Reasoning for Function *callee*, defined in Fig. 1

$$\frac{(\mathcal{N};\Phi_1) \hookrightarrow \Phi_1' \qquad (\mathcal{N};\Phi_2) \hookrightarrow \Phi_2'}{\mathcal{N}\,;\,(\Phi_1 \vee \Phi_2) \hookrightarrow \Phi_1' \vee \Phi_2'}\ [\mathcal{N}\text{-}Norm\text{-}Disj] \qquad \frac{\theta{=}\mathcal{N}';\theta' \qquad \mathcal{N}'' \hookrightarrow \mathcal{N};\mathcal{N}'}{\mathcal{N}\,;\,\theta \hookrightarrow \mathcal{N}''\,;\,\theta'}\ [\mathcal{N}\text{-}Norm\text{-}Flow]$$

$$\frac{\mathcal{N} = \exists x^* \cdot \mathbf{req}\ P\ \mathbf{ens}\ Q \qquad \mathcal{N}' = \exists y^* \cdot \mathbf{req}\ P'\ \mathbf{ens}\ Q' \qquad Q * ?P_A \vdash_{Bi\text{-}ab} P' * ?Q_F \qquad \mathcal{N}'' = \exists x^*, y^* \cdot \mathbf{req}\ P * P_A\ \mathbf{ens}\ Q' * Q_F}{\mathcal{N}\,;\,\mathcal{N}' \hookrightarrow \mathcal{N}''}\ [\mathcal{N}\text{-}Norm\text{-}Norm]$$

Fig. 21. Deployed Normalization Rules.

5.2 Reduction of Try-Catch Constructs

Given any handler type δ, any *ESL* formula Φ, and any handler specification $\mathcal{H}_\Phi$, the relation $(\mathbf{try}[\delta](\Phi)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \Phi')$ holds if after Φ is handled by $\mathcal{H}_\Phi$, the staged flows result to Φ'. All the disjunction within the try-block are reduced independently, by $[\mathcal{R}\text{-}Disj]$, shown as follows:

$$\mathbf{try}[\delta](\Phi_1 \vee \Phi_2)\ \mathbf{catch}\ \mathcal{H}_\Phi = \mathbf{try}[\delta](\Phi_1)\ \mathbf{catch}\ \mathcal{H}_\Phi \vee \mathbf{try}[\delta](\Phi_2)\ \mathbf{catch}\ \mathcal{H}_\Phi \qquad [\mathcal{R}\text{-}Disj]$$

The complete set of try-catch reduction rules is given in Fig. 22. We assume that the specification of the try-block is in a normalized form, i.e., $(\theta = \mathcal{E}^*\,;\,\mathcal{N})$, and the nested handlers are reduced before hand, if possible; otherwise, they would be left as irreducible.

Rule $[\mathcal{R}\text{-}Normal]$ denotes the base case, which is applied when there is a normal stage $\mathcal{N}$ and returns r, indicating the execution of the handled program has finished. In this case, the resulting staged flows are achieved by composing $\mathcal{N}$ with the instantiated specification of the normal clause. This step corresponds to the operational semantics rule (*OP-Ret*) in Fig. 14. Rule $[\mathcal{R}\text{-}Skip]$ is applied when the starting flow is an effect stage $\mathcal{E}$, which the current handler cannot handle, corresponding to (*OP-Unhandled*). In this case, it adds $\mathcal{E}$ into the history and continues to reason about the rest of the flow. Rule $[\mathcal{H}\text{-}Unfold]$ unfolds the non-recursive definition of a higher-order stage. In cases where f is unknown (cf. the example in Sec. 3.3), the try-catch construct is left as irreducible at this moment and the reduction resumes whenever f is suitably instantiated.

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

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ E(x,r) \qquad E \notin dom(\mathcal{H}_\Phi)}{\mathbf{try}[\delta](\mathcal{E}\ ;\ \theta)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{E}\ ;\ \mathbf{try}[\delta](\theta)\ \mathbf{catch}\ \mathcal{H}_\Phi} \quad [\mathcal{R}\text{-}Skip]$$

$$\frac{\mathcal{E} = \mathcal{N}\ ;\ f(x^*,r') \qquad (f(y^*,r) = \Phi_f) \in \mathcal{P}}{\mathbf{try}[\delta](\mathcal{E}\ ;\ \theta)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathbf{try}[\delta](\mathcal{N}\ ;\ \Phi_f[x^*/y^*, r'/r]\ ;\ \theta)\ \mathbf{catch}\ \mathcal{H}_\Phi} \quad [\mathcal{R}\text{-}Unfold]$$

$$\frac{\mathcal{E}{=}\mathcal{N}; E(x,r) \qquad E{\in}dom(\mathcal{H}_\Phi) \qquad (x'{\rightarrow}\Phi_n){\in}\mathcal{H}_\Phi \qquad \Phi = \theta[r_1]\ ;\Phi_n[r_1/x']}{\mathbf{try}[s](\mathcal{E}\ ;\ \theta)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathbf{try}[s](\mathcal{E}\ \#\ \Phi)\ \mathbf{catch}\ \mathcal{H}_\Phi} \quad [\mathcal{R}\text{-}Shallow]$$

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

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

$$\frac{\begin{array}{c}\mathcal{E} = \mathcal{N}\ ;\ f(x^*,r') \qquad (rec\ f(y^*,r) = \Phi_f) \in \mathcal{P} \qquad fst(\Phi_f) \in dom(\mathcal{H}_\Phi) \\ (\mathbf{try}[\delta](f(y^*,r)\ \#\ \Phi)\ \mathbf{catch}\ \mathcal{H}_\Phi \sqsubseteq \Phi_{inv}) \in \mathcal{P}\end{array}}{\mathbf{try}[\delta](\mathcal{E}\ \#\ \Phi_c)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow \mathcal{N}\ ;\ \Phi_{inv}[x^*/y^*, r'/r, \Phi_c/\Phi]} \quad [\mathcal{R}\text{-}Lemma\text{-}App]$$

Fig. 22. Reduction Rules for Try-Catch Constructs

Before actually handling any effects, we first reason about the behaviors of its continuation so that when later needed, we could instantiate the high-order predicate k using the continuation's specification. Thus, we introduce a new intermediate try-catch logic construct of the form $(\mathbf{try}[\delta](\mathcal{E}\ \#\ \Phi_c)\ \mathbf{catch}\ \mathcal{H}_\Phi)$ where $\mathcal{E}$ is the current effect stage, and Φ_c is the reduced specification for $\mathcal{E}$'s continuation with respect to the definition of $\mathcal{H}_\Phi$, formally defined in Fig. 23.

With that, we distinguish the reasoning of the deep and shallow handlers. Rule $[\mathcal{R}\text{-}Shallow]$ handles the continuation using the normal clause of $\mathcal{H}_\Phi$, corresponding to our semantics of shallow handlers (*OP-Shallow*), which are only installed for the first effect and the final normal return, and inserts a # marker between the current effect stage and the handled continuation, i.e., Φ. Whereas

$$\begin{array}{ll}[\checkmark, S, h] \rightsquigarrow_m [C, S_1, h_1, R] \models & \text{iff} \quad \mathcal{E}{=}\mathcal{N}\ ;\ E(x,r) \text{ and } (E(y)k{\rightarrow}\varphi_1) \in \mathcal{H}_\Phi \text{ and} \\ \quad \mathbf{try}[\delta](\mathcal{E}\ \#\ \varphi[r_c])\ \mathbf{catch}\ \mathcal{H}_\Phi & \quad\quad [\checkmark, S, h] \rightsquigarrow_m [C_2, S_2, h_2, Norm(\_)] \models \mathcal{N} \\ & \quad\quad S_3 = S_2{+}[k := \lambda\,(r,r_c){\rightarrow}\,\varphi[r_c]] \text{ and} \\ & \quad\quad [C_2, S_3, h_2] \rightsquigarrow_m [C, S_1, h_1, R] \models \varphi_1[x/y]\end{array}$$

Fig. 23. The Semantics of the # Try-Catch Construct

[$\mathcal{R}$-*Deep*] reasons about the continuation using $\mathcal{H}_\Phi$ unchanged, corresponding to the semantics of deep handlers (*OP-Deep*), which are persistently installed. After the continuations are properly handled, rule [$\mathcal{R}$-*Handle*] handles the current effect by instantiating the arguments and bind k using a lambda specification constructed using the specification for the continuation.

A try-catch lemma of a try-catch construct is in the form $(\mathbf{try}[\delta](f(y^*, r)\,\#\,\Phi)\ \mathbf{catch}\,\mathcal{H}_\Phi\ =\ \Phi_{inv})$ where f is recursively defined and Φ is the specification for the already handled continuation. Rule [$\mathcal{R}$-*Lemma-App*] firstly retrieves the lemma definition and reduces the current try-catch construct to the instantiated summary, i.e., $\Phi_{inv}[x^*/y^*, r'/r, \Phi_c/\Phi]$. Note that the side condition requires that the first possible effect generated by f must be handled by the current handler, expressed as "$fst(\Phi_f) \in dom(\mathcal{H}_\Phi)$". Otherwise, [$\mathcal{R}$-*Unfold*] and [$\mathcal{R}$-*Skip*] shall be applied before hand.

Theorem 5.1 defines the soundness of the try-catch reduction process. Moreover, the reduction always terminates successfully if sufficient lemmas are given and proven; otherwise, it can be left as irreducible when there are unknown predicates or lemmas. Termination of the try-catch reduction rules themseves can be proven with the help of the following decreasing measure.

$$
\begin{array}{rcl}
M[\mathbf{try}(\Phi_1 \# \Phi_2)\mathbf{catch}\,H] & < & M[\mathbf{try}(\Phi_1; \Phi_3)\mathbf{catch}\,H] \\
M[\mathbf{try}(\Phi_2)\mathbf{catch}\,H] & < & M[\mathbf{try}(\Phi_1; \Phi_2)\mathbf{catch}\,H] \\
M[\mathbf{try}(\Phi_1; \mathrm{unfold}(f(x^*)); \Phi_2)\mathbf{catch}\,H] & < & M[\mathbf{try}(\Phi_1; f(x^*); \Phi_2)\mathbf{catch}\,H], \text{if } f \text{ is non-recursive} \\
M[\Phi_1] & < & M[\Phi_2], \text{if } \Phi_1 \text{ is a sub-term of } \Phi_2
\end{array}
$$

Discussion. Note that the induction principle applies to both deep and shallow handlers in a unified manner, which supports arbitrary recursive calls *without* restricting to one-shot continuations. We demonstrate the applicability using a left recursive toss function in both deep and shallow handlers in Appendix A [TR 2024]. To the author's knowledge, this cannot be achieved by prior works.

5.3 Soundness Proofs

Theorem 5.1 (Soundness of Reduction)**.** *Given any try-catch reduction,* $\mathbf{try}[\delta](\Phi)\ \mathbf{catch}\,\mathcal{H}_\Phi \rightsquigarrow \Phi'$*; forall* $S, h, C_1, S_1, h_1,$ *and* R_1*, if* $[\checkmark, S, h] \rightsquigarrow_m [C_1, S_1, h_1, R_1] \models \mathbf{try}[\delta](\Phi)\ \mathbf{catch}\,\mathcal{H}_\Phi$*, then* $[\checkmark, S, h] \rightsquigarrow_m [C_1, S_1, h_1, R_1] \models \Phi'$ *holds.*

Proof. By induction on the structure of the reduction rule, elaborated in Appendix B.1 [TR 2024]. □

Theorem 5.2 (Soundness of Normalization)**.** *Given any normalization rule, if it concludes that* $(\varphi \hookrightarrow \Phi)$*; forall* $S_0, h_0, C_1, S_1, h_1,$ *and* R_1*, if* $[\checkmark, S_0, h_0] \rightsquigarrow_m [C_1, S_1, h_1, R_1] \models \varphi$*, then* $[\checkmark, S_0, h_0] \rightsquigarrow_m [C_1, S_1, h_1, R_1] \models \Phi$ *holds.*

Proof. By induction on the structure of the normalization rule, elaborated in Appendix B.2 [TR 2024]. □

Theorem 5.3 (Soundness of Forward Rules)**.** *If the forward reasoning concludes that a given triple* $\vdash \{\mathcal{N}\}\ e\ \{\Phi\}$ *is valid, forall* S_0 *and* h_0*, if* $[\checkmark, S_0, h_0] \rightsquigarrow_m [\checkmark, S, h, Norm(\_)] \models \mathcal{N}$*, and let* $SH = \{(S_1, h_1, R_1) \mid [\checkmark, S_0, h_0] \rightsquigarrow_m [\checkmark, S_1, h_1, R_1] \models \Phi\}$*, and* $SH \neq \emptyset$*, then* $[S, h, e] \longrightarrow [S_2, h_2, R_2] \wedge R_2 \neq Err \wedge \exists (S_3, h_3, R_3) \in SH \cdot ([\checkmark, S_0, h_0] \rightsquigarrow_m [\checkmark, S_3, h_3, R_3] \models \Phi \wedge S_3 \supseteq S_2,\ h_3 = h_2,\ R_2 \in R_3)$*.*

Proof. By induction on the derivation of $[S, h, e] \longrightarrow [S_2, h_2, R_2]$, and the proofs are built on top of Theorem 5.1 and Theorem 5.2. Details are elaborated in Appendix B.3 [TR 2024]. □

To understand Theorem 5.3, recall that in the triple $\vdash \{\mathcal{N}\}\ e\ \{\Phi\}$, Φ describes the behavior of e if it is executed in some state *following* some "historical" execution that ends with $\mathcal{N}$. During the verification, as e is traversed, the forward verification rules intuitively transform the *history*, resulting in a final abstraction of its behavior denoted as Φ. In the statement of the theorem, S_0, h_0 is the initial state prior to the execution of the history $\mathcal{N}$, which leads to the state S, h in which e will be executed, in turn leading to the state S_2, h_2, R_2. The theorem says that given a triple $\vdash \{\mathcal{N}\}\ e\ \{\Phi\}$ whose validity is witnessed by the non-emptiness of the set SH (which contains the safe final states S_1, h_1, R_1 satisfying Φ, given initial states S_0, h_0), then the execution of e will result in a (safe) state corresponding to some state S_3, h_3, R_3 in SH. This correspondence is, concretely, that the final heap should be identical, the final result should be non-erroneous and identical, and the store of Φ should be a superset of that of e (due to variables arising from staged existentials).

6 ENTAILMENT CHECKING

Given two *ESL* formulae Φ_1 and Φ_2, this section presents an algorithm for automatically checking the entailment relation $\Phi_1 \sqsubseteq \Phi_2$. Intuitively, proving $\Phi_1 \sqsubseteq \Phi_2$ amounts to checking whether all the possible flows in the antecedent Φ_1 form a subset of all the possible flows in the consequent Φ_2.

$$\frac{P_A \vdash \Phi_1 \sqsubseteq \Phi \rightsquigarrow Q_F^1 \qquad P_A \vdash \Phi_2 \sqsubseteq \Phi \rightsquigarrow Q_F^2}{P_A \vdash \Phi_1 \vee \Phi_2 \sqsubseteq \Phi \rightsquigarrow Q_F^1 \vee Q_F^2}\ [\textit{Entail-LHS-OR}] \qquad \frac{P_A \vdash \theta \sqsubseteq \Phi_i \rightsquigarrow Q_F^i \qquad i \in \{1,2\}}{P_A \vdash \theta \sqsubseteq \Phi_1 \vee \Phi_2 \rightsquigarrow Q_F^i}\ [\textit{Entail-RHS-OR}]$$

$$\frac{g(x^*, r) = \Phi_1 \in \mathcal{P} \qquad P_A \vdash \mathcal{N}_1\ ;\ \Phi_1[y^*/x^*][r_1/r] \sqsubseteq \Phi \rightsquigarrow Q_F}{P_A \vdash \mathcal{N}_1\ ;\ g(y^*, r_1) \sqsubseteq \Phi \rightsquigarrow Q_F}\ [\textit{Entail-Unfold-LHS}] \qquad \frac{P_A \vdash \mathcal{E}_1 \sqsubseteq \mathcal{E}_2 \rightsquigarrow Q_F^1 \qquad Q_F^1 \vdash \theta_1 \sqsubseteq \theta_1 \rightsquigarrow Q_F^2}{P_A \vdash (\mathcal{E}_1\ ;\ \theta_1) \sqsubseteq (\mathcal{E}_2\ ;\ \theta_2) \rightsquigarrow Q_F^2}\ [\textit{Entail-Flow}]$$

$$\frac{P_A * P_2 \vdash_{sl} (\exists x^* \cdot P_1) \rightsquigarrow Q_F^1 \qquad Q_F^1 * Q_1 \vdash_{sl} (\exists y^* \cdot Q_2) \rightsquigarrow Q_F^2}{P_A \vdash (\exists\, x^* \cdot \mathbf{req}\ P_1\ \mathbf{ens}\ Q_1) \sqsubseteq (\exists\, y^* \cdot \mathbf{req}\ P_2\ \mathbf{ens}\ Q_2) \rightsquigarrow Q_F^2}\ [\textit{Entail-Norm}]$$

$$\frac{P_A \vdash \mathcal{N}_1 \sqsubseteq \mathcal{N}_2 \rightsquigarrow Q_F^1 \qquad Q_F^1 \vdash_{sl} x{=}y \wedge r_1{=}r_2 \rightsquigarrow Q_F^2}{P_A \vdash \mathcal{N}_1\ ;\ E(x, r_1) \sqsubseteq \mathcal{N}_2\ ;\ E(y, r_2) \rightsquigarrow Q_F^2}\ [\textit{Entail-Eff}]$$

$$\frac{P_A \vdash \mathcal{N}_1 \sqsubseteq \mathcal{N}_2 \rightsquigarrow Q_F^1 \qquad Q_F^1 \vdash_{sl} x^*{=}y^* \wedge r_1{=}r_2 \rightsquigarrow Q_F^2}{P_A \vdash \mathcal{N}_1\ ;\ g(x^*, r_1) \sqsubseteq \mathcal{N}_2\ ;\ g(y^*, r_2) \rightsquigarrow Q_F^3}\ [\textit{Entail-HO}]$$

$$\frac{P_A \vdash \mathcal{N}_1 \sqsubseteq \mathcal{N}_2 \rightsquigarrow Q_F^1 \qquad \delta_1{=}\delta_2 \qquad Q_F^1 \vdash \Phi_1 \sqsubseteq \Phi_2 \rightsquigarrow Q_F^2}{P_A \vdash \mathcal{N}_1\ ;\ \mathbf{try}[\delta_1](\Phi_1)\ \mathbf{catch}\ \mathcal{H}_\Phi \sqsubseteq \mathcal{N}_2\ ;\ \mathbf{try}[\delta_2](\Phi_2)\ \mathbf{catch}\ \mathcal{H}_\Phi \rightsquigarrow Q_F^2}\ [\textit{Entail-Try-Catch}]$$

Fig. 24. Selected Entailment Rules for Normalized Staged Flows

Due to the choice of separation logic as a base logic, the entailments between two staged flows are of the form $P_A \vdash \Phi_1 \sqsubseteq \Phi_2 \rightsquigarrow Q_F$, where P_A is a given assumption (initialized by $emp \wedge true$) and Q_F is the residue (as the result of frame inference), both of which are state formulae. The entailment rules

are shown in Fig. 24. Rules [*Entail-LHS-OR*] and [*Entail-RHS-OR*] handle disjunctive antecedents and consequents, respectively. Every flow in the antecedent must be allowed by the consequent, whereas it is acceptable to have additional flows in the consequent that do not correspond to flows in the antecedent. The rest of the rules are for entailments between two disjunction-free staged flows. In particular, the rule [*Entail-Flow*] demonstrates how the use of normalized specifications "aligns" the antecedent and consequent, allowing entailment proofs to be carried out stage by stage, starting with the heads of both flows. [*Entail-Unfold-LHS*] allows the use of provided nonrecursive predicate definitions; there is an analogous rule for unfolding on the right. Recursive predicate definitions are handled via lemmas, in the same manner as try-catch formulae. The proving of lemmas is based on the cyclic proof principles [Brotherston 2005], which rewrites the formulae by taking turns applying the unfolding rules and try-catch reduction rules.

Rule [*Entail-Norm*] handles normal stages, which are pre/post specifications, and this reduces to checking the contra-variance of preconditions and covariance of postconditions. Separation logic proof obligations, i.e., $\vdash_{sl}$, are reduced to decidable first-order theory that fits well into the satisfiability modulo theories (SMT) framework, which is standard [Chin et al. 2011; Piskac et al. 2013]. The following rules are for different cases regarding the suffixes of given stages $\mathcal{E}$.

Effect stages and predicate stages whose definitions are unknown are treated similarly, as shown in the rules [*Entail-Eff*] and [*Entail-HO*]. When proving the intermediate stages, the symbols, i.e., E or g, used have to match and under the given assumptions P_A and the frame produced by the preceding normal stage Q_F^1, the formal arguments and return variables must be provably equal terms. In the rule [*Entail-Try-Catch*], try-catch stages must match more or less exactly. Not much is done here as they are best eliminated away by the try-catch reduction; when this is not possible, there is often no better choice than to leave them in specifications.

The entailment checking is terminating because the length of staged flows and the number of disjunctions in Φ are considered finite. The soundness of the entailment checking is defined in Theorem 6.1, making use of a *model set relation*, defined in Definition 6.1, which abstracts the set of final states that one given *ESL* specification Φ can accept.

Definition 6.1 (Model Set Relation)**.** *Given any S, h, SH, and Φ, we say they have the relation of $([\checkmark, S, h] \rightsquigarrow_m^{set} SH \models \Phi)$, iff $SH = \{(C_1, S_1, h_1, R_1) \mid [\checkmark, S, h] \rightsquigarrow_m [C_1, S_1, h_1, R_1] \models \Phi\}$ holds.*

Theorem 6.1 (Soundness of Entailment Rules)**.** *If the entailment checking proves that an entailment $(P_A \vdash \Phi_1 \sqsubseteq \Phi_2 \rightsquigarrow Q_F)$ is valid, forall S, h, SH_1 and SH_2, if $([\checkmark, S, h] \rightsquigarrow_m^{set} SH_1 \models \Phi_1 \wedge P_A)$ and $([\checkmark, S, h] \rightsquigarrow_m^{set} SH_2 \models \Phi_2 \wedge Q_F)$, then $SH_1 \subseteq SH_2$ holds.*

Proof. By induction on the structure of the entailment rule, elaborated in Appendix B.4 [TR 2024]. □

7 ANOTHER CASE STUDY AND EXPERIMENTAL RESULTS

Apart from proving the soundness of our approach, we prototype an automated verification tool, Heifer$^+$, for 5K LoC on top of OCaml 5, targeting OCaml programs with the effect syntax from Multicore OCaml[2]. Here, we demonstrate one more case study and present experimental results based on a suite of benchmark programs. Experiments were done on a MacBook with a 2.6 GHz 6-Core Intel i7 processor. The source code and the evaluation benchmark are openly accessible [Zenodo 2024].

[2] https://github.com/ocaml-multicore/ocaml-multicore

7.1 Case Study: McCarthy's angelic choice operator

McCarthy's ambiguous operator *amb*[3] is an interesting mathematical operator that can rewind into the past whenever it sees trouble and try a different choice. Describing its semantics using multi-shot continuation is much less complex, as shown in Fig. 25. The *amb* function takes a list of boolean values *xs*, and its handler iterates the list and resumes the continuation with each boolean element. For simplicity, we used a list of booleans instead of a list of thunks. If (*Failure* 500) is raised from the continuation, the handler omits the exception and continues the iteration. If any continuation succeeds, it invokes a *Success* effect, which will be caught, re-raised, and caught again; finally, it handles the *Success* effect by returning its carried value. To make the example more challenging, we added a mutable counter to record how many times the iterator had backtracked before reaching the first succeeding element.

```
effect Choose : bool list -> bool
effect Success : int -> unit
effect Failure : int -> int

let amb (xs:bool list) counter : bool
= let b = perform (Choose xs) in counter := !counter +1; b

let f xs counter = if amb xs counter then 7 else perform (Failure 500)

let handle (xs : bool list) counter : int
= match (f xs counter) with
| x -> x
| effect (Choose xs) k ->
  match List.iter (fun ele ->
   match let seven = resume k ele in perform (Success seven) with
   | effect (Success x) k -> perform (Success x)
   | effect (Failure _) k -> () (* Omitting Failure 500 *)
   | _ -> ()
   ) xs; (* iterate the lambda elements from xs *)
   perform (Failure 404)
  with | x -> x | effect (Success r) k -> r (* Leaking Failure 404 *)
```

Fig. 25. McCarthy's Locally Angelic Choice Operator

This example is non-trivial as it involves nested handlers with higher-order functions; heap-manipulating behaviors in the continuation; performing effects while handling effects; encoding exceptions using effects; and last but not least, reasoning is also required on the list data structure.

We present the specifications for *handle* and the key predicate definitions in Fig. 26. Recursively defined predicate *containRetTrue* takes a list *xs*, uses a Boolean variable *r* to denote if there exists an element from the list *xs* which equals to *true*, and uses an integer *p* to denote the position of such an element. In case there isn't such an element, *p* equals the length of the list. The specification for *handle* indicates that if there exists one element in a given list *xs*, that equals to *true*, then it returns 7, and the counter is increased by the position of the value, *p*; otherwise, it carries an unhandled effects (*Failure* 404) as the final result. From the specification, we can see that the (*Failure*

[3] https://okmij.org/ftp/ML/ML.html#amb

$$
\begin{aligned}
\mathit{containRetTrue}(xs, p, r) &= \mathbf{ens}[r]\ xs{=}[] \wedge p{=}0 \wedge r{=}\mathit{false} \\
&\quad \vee \exists h, t \cdot \mathbf{ens}[r]\ xs{=}h{::}t \wedge h{=}\mathit{true} \wedge p{=}1 \wedge r{=}\mathit{true} \\
&\quad \vee \exists h, t, p' \cdot \mathbf{ens}[r]\ xs{=}h{::}t \wedge h{=}\mathit{false} \wedge p{=}p'{+}1 \wedge \mathit{containRetTrue}(t, p', r) \\
\mathit{List.iter}(f, xs, r) &= \mathbf{ens}[r]\ xs{=}[] \wedge r{=}() \ \vee\ \exists h, t \cdot \mathbf{ens}\ xs{=}h{::}t\,;\ f(h, ())\,;\ \mathit{List.iter}(f, t, r) \\
\mathit{amb}(xs, \mathit{counter}, r) &= \exists b \cdot \mathit{Choose}(xs, b)\,;\ \exists z \cdot \mathbf{req}\ \mathit{counter}{\mapsto}z\ \mathbf{ens}[r]\ \mathit{counter}{\mapsto}z{+}1 \wedge r{=}b \\
f(xs, \mathit{counter}, r) &= \exists b \cdot \mathit{amb}(xs, \mathit{counter}, b)\,;\ (\mathbf{ens}[r]\ b{=}\mathit{true} \wedge r{=}7 \ \vee\ \mathbf{ens}\ b{=}\mathit{false}\,;\ \mathit{Failure}(500, r)) \\
\mathit{handle}(xs, \mathit{counter}, r) &= \exists z, p, b \cdot \mathbf{req}\ \mathit{counter}{\mapsto}z \wedge \mathit{containRetTrue}(xs, p, b)\,; \\
&\quad (\mathbf{ens}[r]\ \mathit{counter}{\mapsto}z{+}p \wedge b{=}\mathit{true} \wedge r{=}7 \ \vee\ \mathbf{ens}\ \mathit{counter}{\mapsto}z{+}p \wedge b{=}\mathit{false}\,;\ \mathit{Failure}(404, r))
\end{aligned}
$$

Fig. 26. Specification for the *handle* Function and Deployed Predicate definitions

500) effects possibly performed by f – indicate the failure of individual attempts – will be omitted by the handler at line 17, whereas (*Failure* 404) effects performed *in the handler* at line 20 may escape the *handle* function, to indicate that there is no element in *xs* which equals to true.

7.2 Experimental Results

The benchmark programs in Table 1 are based on the examples from various sources [4]. More specifically, program 1 is taken and revised from a *Memory Cell with Exchange* example in the prior work [de Vilhena and Pottier 2021], to further model different operations with a state monad, including read, write, exchange values, and applications of their combinations. Programs 3 and 7 are originally from the *multicont* library, which provides practical examples for multi-shot continuations. Program 5 revises program 3 by changing the handler to be shallow. Programs 4 and 6 demonstrate the usages of lemmas for left recursive functions in both deep and shallow handlers. Program 2 is newly created in this work for feature diversity.

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

In Table 1, columns **LoC** and **LoS** record the lines of code and lines of specification, respectively. The column **Total** sums up the time for forward reasoning and entailment checking. The column **AskZ3** records the time consumed by the Z3 solver during the whole verification process. Times are measured in seconds. Although forward reasoning and entailment checking are mostly automated, we show that the verification is non-trivial by labeling the programs with features: **Ind** indicates

[4] Mainly https://github.com/dhil/ocaml-multicont and https://v2.ocaml.org/manual/effects.html.

whether the proof is inductive, **MultiS** means if the program uses multi-shot handlers, **ImpureC** indicates if there exist heap-manipulation operations in the continuations. Lastly, **HO** indicates if the program is higher-order, i.e., function inputs or outputs are of function type.

As shown, for 461 lines of code in total, our verification system intakes 129 lines of specifications, with an average LoS/LoC ratio of 28%. For the time consumption, majority of the time is consumed by the Z3 solving [5] – taking up 68.8% (23.32/33.88) of the total verification time. To the best of the authors' knowledge, this work is the first that provides verification solutions for the given benchmark and supports automated proofs, demonstrating the feasibility of *ESL*. Due to space limitation, we demonstrate the verification for programs 1, 2, 4, 5, and 6 in Appendix C [TR 2024].

8 RELATED WORK

In this section, we discuss related works on the reasoning of algebraic effects and delimited control, applications for multishot continuations, and other structured specifications.

Reasoning about algebraic effects and delimited control. For decades, monads [Moggi 1989; Wadler 1990] have dominated the scene of functional programming with effects. The recent popularization of algebraic effects and handlers [Bauer and Pretnar 2015; Plotkin and Pretnar 2009] promises to change the landscape. To support resumption, an effect handler has access to the continuation at the point of effect invocation. Thus, algebraic effects and handlers provide a form of delimited control operators, which have long been used to encode effects [Danvy 2006]. In monads, the effectful behavior is defined in *bind* and *return*, statically determining the behavior inside the *do* block. Whereas the behaviors of performing algebraic effects are determined dynamically by the encompassing handlers, which gives greater flexibility in the composition of effectful code, but it also requires additional reasoning to regulate the composed behaviors.

Many prior works study the semantics of effects and effect handlers in a pure setting. For instance, Plotkin and Pretnar [2008] propose a logic to reason about the equality of computations in a calculus with effects but no handlers. They later introduce effect handlers as an internal way of giving meaning to effects [Plotkin and Pretnar 2013] and discuss a notion of correctness whereby a handler is correct if it satisfies an intended equational theory. In addition, there is also a bulk of work on temporal verification for algebraic effects. For example, Gordon [2020] are concerned with infinite-state higher-order programs with control operators using type and effect systems, and it defines a framework for sequential effects with tagged control operators akin to abort and call/cc, capturing temporal safety properties. Song et al. [2022] propose a trace-based logic for practical automated temporal verification of effect handlers. Closely related are verification works for delimited control operators and properties beyond the safety and liveness of individual programs. Kiselyov et al. [2021] recover equational reasoning in the presence of effect handlers. Sekiyama and Unno [2023] present a type-and-effect system for $shift_0/reset_0$ [Materzok and Biernacki 2011] which reasons about the traces that continuations generate, supporting liveness reasoning. Afterward, a follow-up work [Kawamata et al. 2024] proposes a refinement type system for languages with algebraic effects and handlers. These works typically avoid heap-manipulating behaviors. To support heap-based programs, Delbianco and Nanevski [2013] propose HTT_{cc}, a separation logic for calculus with dynamically allocated mutable state and an *algebraic* variant of call/cc and throw, but did not provide support for delimited continuations. Their solution also uses "large footprint" assertions, and does not have the frame rule, which means that it imposes reasoning about the entire heap.

[5] For better precision, some pre-analysis and encoding are carefully designed, which query Z3 additionally to incorporate the multiplication operator in program 2 and the power operator used in programs 3-4.

The current state-of-the-art for verifying imperative behaviors in algebraic effects and handles is [de Vilhena and Pottier 2021]. It specifies program behaviors using separation logic and client-handler interactions in the form of *protocols*, which globally define the effects that clients may perform and the replies they may receive from handlers. A similar protocol-based solution is used in Cameleer [Soares and Pereira 2023], which encodes effects as WhyML exceptions and uses defunctionalization to represent higher-order continuations. However, these solutions are restricted to one-shot continuations because those protocols are incapable of accessing and composing the continuations explicitly. Subsequently, de Vilhena [2022] extends de Vilhena and Pottier [2021] to multi-shot resumption, restrictively allowing only persistent assertions in the continuations of effects. The extension, however, still precludes multi-shot impure continuations, e.g., the examples in Sec. 3.1 and Fig. 5. This key limitation motivates our work, and we propose *ESL* to fill this gap by naturally supporting unrestricted handlers with heap-based continuations. As a final note, protocol-based solution uses global assumptions on handlers to provide explicit (or early) interpretation for each of the algebraic effects. In theory, this explicit approach could help simplify the reasoning needed for algebraic effects. However, in reality, it also leads to restrictions on the types of continuations that can be supported for multi-shot handlers, and moreover loses some flow-sensitivity that is inherent in try-catch handling of effects with continuations. In contrast to protocol-based approach, our solution models try-catch handlers *separately* from effect invocations, and *delays the interpretation* of the algebraic effects until try-catch handlers are defined and where the scope of the continuation(s) would also become known. It would be interesting to explore how the benefits from both these approaches could be combined.

Applications of multishot continuations. Most effects may be implemented using one-shot continuations [Bruggeman et al. 1996]. Two exceptions are i) to implement a Unix-style asynchronous *fork* primitive [Leijen 2017]; and ii) nondeterminism. The latter is useful in backtracking, probabilistic programming [Nguyen et al. 2022][6], and model checking[7]; as multi-shot handlers naturally express the branching points in search. While backtracking may be expressed by lifting client programs to asymmetric coroutines [de Moura and Ierusalimschy 2009] (closely related to one-shot continuations [Kawahara and Kameyama 2020]), this approach is not modular and does not allow for the creation of abstractions, as effects do. Moreover, nondeterminism may be implemented by throwing exceptions and saving enough state to replay to branching points [Koppel et al. 2018], but this approach incurs runtime overhead from repeated computations, and it assumes the absence of side effects. In contrast, the main advantages of multi-shot continuations are that they simplify the implementation of complex algorithms and generally perform better by incurring more memory overheads instead.

Verifying Higher-Order Imperative Programs. An early approach for reasoning about higher-order imperative programs in pre/post Hoare logic was proposed by Honda et al. [Yoshida et al. 2007]. Our work drew early inspiration from this line of work, and have now extended it to take into account algebraic effects, try-catch handlers and continuations. Our current solution is also built on top of the staged specification mechanism [Foo et al. 2023, 2024] that we have recently proposed to support formal reasoning of imperative higher-order programs. With this new work, we have now extended it to handle algebraic effects, showed how delimited continuations can be captured via staged flows and modelled try-catch logic reduction in a general but precise manner.

[6] https://github.com/Arnhav-Datar/EffPPL
[7] https://github.com/ocaml-multicore/dscheck

9 CONCLUSION

This work is mainly motivated by *how to modularly specify and verify heap-manipulating programs with multi-shot effect handlers*. We present an *Effectful Specification Logic* to write compact and generic specifications for target programs. Our contributions are manifold: we define the forward reasoning rules for a core language, we devise a set of reduction rules to calculate the heap-manipulating behaviors for effect handlers, where zero-/one-/multi-shot continuations coexist. We prototype and automate the proposal, present experimental results, and demonstrate nontrivial and practical case studies to show feasibility. To the best of the authors' knowledge, this work is the first that lays the foundation for a practical verification framework that is capable of modeling arbitrary imperative higher-order programs with algebraic effects and unrestricted handlers.

DATA AVAILABILITY

The source code of the tool, the dataset, and the appendix are available from [Zenodo 2024].

ACKNOWLEDGMENTS

This research is supported by the Ministry of Education, Singapore, under its MOE Academic Research Fund Tier 3 (RIE2025) (MOE Award No: MOET-MOET32021-0001), and by the Ministry of Education, Singapore, under the Academic Research Fund Tier 1 (FY2023) (Project Title: Automated Verification for Imperative Higher-Order Programs). We thank anonymous reviewers for their insightful comments, which led to improvements in the paper's presentation. Thanks also to Paulo and François for clarifying an aspect of their work.

REFERENCES

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001

James Brotherston. 2005. Cyclic proofs for first-order logic with inductive definitions. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 78–92.

Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing control in the presence of one-shot continuations. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 289–300. https://doi.org/10.1145/1480881.1480917

Wei-Ngan Chin, Cristina David, and Cristian Gherghina. 2011. A HIP and SLEEK verification system. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 9–10. https://doi.org/10.1145/2048147.2048152

Olivier Danvy. 2006. An analytical approach to program as data objects.

Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31 (2009), 6:1–6:31.

Paulo Emílio de Vilhena. 2022. *Proof of Programs with Effect Handlers*. Ph. D. Dissertation. Université Paris Cité.

Paulo Emílio de Vilhena and François Pottier. 2021. A separation logic for effect handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434314

Germán Andrés Delbianco and Aleksandar Nanevski. 2013. Hoare-style reasoning with (algebraic) continuations. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 363–376. https://doi.org/10.1145/2500365.2500593

Darius Foo, Yahui Song, and Wei-Ngan Chin. 2023. *Staged Specifications for Automated Verification of Higher-Order Imperative Programs*. Technical Report. National University of Singapore. DOI:arxiv-2308.00988.

Darius Foo, Yahui Song, and Wei-Ngan Chin. 2024. Staged Specifications for Automated Verification of Higher-Order Imperative Programs. In *FM 2024: Formal Methods - 26th International Symposium on Formal Methods, Milan, Italy, Sept 9-13, 2024. Proceedings (Lecture Notes in Computer Science)*. Springer.

Colin S. Gordon. 2020. Lifting Sequential Effects to Control Operators. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:30. https://doi.org/10.4230/LIPICS.ECOOP.2020.23

Daniel Hillerström and Sam Lindley. 2018. Shallow effect handlers. In *Asian Symposium on Programming Languages and Systems*. Springer, 415–435.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. *ACM SIGPLAN Notices* 48, 9 (2013), 145–158.

Satoru Kawahara and Yukiyoshi Kameyama. 2020. One-Shot Algebraic Effects as Coroutines. In *Trends in Functional Programming - 21st International Symposium, TFP 2020, Krakow, Poland, February 13-14, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12222)*, Aleksander Byrski and John Hughes (Eds.). Springer, 159–179. https://doi.org/10.1007/978-3-030-57761-2_8

Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. 2024. Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 8, POPL (2024), 115–147. https://doi.org/10.1145/3633280

Oleg Kiselyov, Shin-Cheng Mu, and Amr Sabry. 2021. Not by equations alone. *Journal of Functional Programming* 31 (2021).

James Koppel, Gabriel Scherer, and Armando Solar-Lezama. 2018. Capturing the future by replaying the past (functional pearl). *Proceedings of the ACM on Programming Languages* 2 (2018), 1 – 29.

Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. 2014. Shape Analysis via Second-Order Bi-Abduction. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 52–68. https://doi.org/10.1007/978-3-319-08867-9_4

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

Daan Leijen. 2017. Structured asynchrony with algebraic effects. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, Sam Lindley and Brent A. Yorgey (Eds.). ACM, 16–29. https://doi.org/10.1145/3122975.3122977

Marek Materzok and Dariusz Biernacki. 2011. Subtyping delimited continuations. In *Proceeding of the 16th ACM SIGPLAN intrernational conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 81–93. https://doi.org/10.1145/2034773.2034786

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 14–23. https://doi.org/10.1109/LICS.1989.39155

Minh Nguyen, Roly Perera, Meng Wang, and Nicolas Wu. 2022. Modular probabilistic models via algebraic effects. *Proc. ACM Program. Lang.* 6, ICFP (2022), 381–410. https://doi.org/10.1145/3547635

Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, K. C. Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *CoRR* abs/2308.08347 (2023). https://doi.org/10.48550/ARXIV.2308.08347 arXiv:2308.08347

Ruzica Piskac, Thomas Wies, and Damien Zufferey. 2013. Automating Separation Logic Using SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 773–789. https://doi.org/10.1007/978-3-642-39799-8_54

Gordon D. Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 118–129. https://doi.org/10.1109/LICS.2008.45

Gordon D. Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5502)*, Giuseppe Castagna (Ed.). Springer, 80–94. https://doi.org/10.1007/978-3-642-00590-9_7

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Log. Methods Comput. Sci.* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL (2023), 2079–2110. https://doi.org/10.1145/3571264

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 206–221.

https://doi.org/10.1145/3453483.3454039

Tiago Soares and Mário Pereira. 2023. A Framework for the Automated Verification of Algebraic Effects and Handlers (extended version). *ArXiv* abs/2302.01265 (2023). https://api.semanticscholar.org/CorpusID:256503855

Yahui Song, Darius Foo, and Wei-Ngan Chin. 2022. Automated Temporal Verification for Algebraic Effects. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 88–109. https://doi.org/10.1007/978-3-031-21037-2_5

Amin Timany and Lars Birkedal. 2019. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.* 3, ICFP (2019), 105:1–105:28. https://doi.org/10.1145/3341709

TR. 2024. Technical Report. https://github.com/songyahui/AlgebraicEffect/blob/StagedSL/docs/ICFP2024_TR.pdf.

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, Gilles Kahn (Ed.). ACM, 61–78. https://doi.org/10.1145/91556.91592

Nobuko Yoshida, Kohei Honda, and Martin Berger. 2007. Logical Reasoning for Higher-Order Functions with Local State. In *Foundations of Software Science and Computational Structures, 10th International Conference, FOSSACS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4423)*, Helmut Seidl (Ed.). Springer, 361–377. https://doi.org/10.1007/978-3-540-71389-0_26

Zenodo. 2024. Benchmark and Source Code. https://doi.org/10.5281/zenodo.11363460.

Received 2024-02-28; accepted 2024-06-18