Automated Temporal Verification for Algebraic Effects

Yahui Song(✉), Darius Foo, and Wei-Ngan Chin

School of Computing, National University of Singapore, Singapore, Singapore
{yahuis,dariusf,chinwn}@comp.nus.edu.sg

Abstract. Although effect handlers offer a versatile abstraction for user-defined effects, they produce complex and less restricted execution traces due to the composable non-local control flow mechanisms. This paper is interested in the temporal behaviors of effect sequences, such as unhandled effects, termination of the communication, safety, fairness, etc. Specifically, we propose a novel effects logic *ContEffs*, to write precise and modular specifications for programs in the presence of user-defined effect handlers and primitive effects. As a second contribution, we devise a forward verifier together with a fixpoint calculator to infer the behaviors of such programs. Lastly, our automated verification framework provides a purely algebraic *term-rewriting system* (TRS) as the back-end solver, efficiently checking the entailments between *ContEffs* assertions. To demonstrate the feasibility, we prototype a verification system where zero-shot, one-shot, and multi-shot continuations coexist; prove its correctness; present experimental results; and report on case studies.

1 Introduction

User-defined effects and effect handlers are advertised and advocated as a relatively easy-to-understand and modular approach to 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. Unlike the exception handlers, the key difference is that effects handlers have access to a continuation. By invoking this continuation, the handler can communicate a reply to the suspended computation and resume its execution.

For example, `effect Yield : int -> unit`, declares the `Yield` effect, to be used in the *generator functions*. When it is performed, the program suspends its current execution and returns the yielded `int` value to the handler. Such usages separate the logic, e.g., iterating a list, from the effectful operations, such as "printing on the console" or "sending an element to a consumer", thereby improving code reuse and memory efficiency. Functions perform effects without needing to know how the handlers are implemented, and the computation may be enclosed by different handlers that handle the same effect differently.

Recently, effect handlers are found in several research programming languages, such as Eff [1], Frank [2], Links [3], Multicore OCaml [4], and Scala [5],

I. Sergey (Ed.): APLAS 2022, LNCS 13658, pp. 88–109, 2022.
https://doi.org/10.1007/978-3-031-21037-2_5

etc. There is a growing need for programmers and researchers to reason about the combination of primitive effects and user-defined handlers. In particular, we are interested in the techniques for inferring and verifying temporal behaviors of such non-local control flows, which have not been extensively studied. In this paper, we tackle the following verification challenges:
1. The coexistence of zero-shot, one-shot and multi-shot continuations. The design decisions of various implementations [4,6] and verification solutions [7,8] diverge upon the question that, should it be permitted or forbidden to invoke a captured continuation more than once? In this paper, our forward inference rules shows the generality to incorporate both one-shot and multi-shot continuations. Furthermore, it naturally supports reasoning on exceptions by treating them as *zero-shot*, i.e., that abandon the continuations completely.
2. Non-terminating behaviors. Figure 1 presents the so-called "recursive cow" program drawn from the benchmark [9], which looks like it is terminating but it actually cycles. Function `f()` performs the predefined effect `Foo`; then `loop ()` handles effect `Foo` by resuming a closure which in turn performs `Foo` when applied.

With higher-order effect signatures and in the setting of deep handlers[1], the communications between the computation and handlers potentially lead to infinite traces. It is useful yet challenging to automatically infer/verify the termination of the communication. In this paper, we devise *ContEffs*, i.e., extended regular expressions with arithmetic constraints, to provide more precise specifications by integrating: $\star$ for finite traces; ω for infinite traces; ∞ for possibly finite or infinite traces.

```
effect Foo : (unit -> unit)

let f() = perform Foo ()

let loop()
= match f () with
| _ -> () (*normal return*)
| effect Foo k -> continue k
  (fun () -> perform Foo ())
```

Fig. 1. A loop.

3. Linear temporal properties. For decades monads have dominated the scene of pure functional programming with effects, and the recent popularization of algebraic effects and handlers promises to change the landscape. However, with rapid change also comes confusion. In monads, the effectful behavior is defined in *bind* and *return*, statically determining the behavior inside the *do* block. Whereas algebraic effects call effectful operations with no inherent behavior. Instead, the behavior is determined dynamically by the encompassing handler. Although this gives greater flexibility in the composition of effectful code, it requires further specifications and verification to enforce the temporal requirements.

In this work, *ContEffs* smoothly encode and go beyond the linear temporal logic (LTL). For examples: *"Effect A will never be followed by effect B"* is a fairness property, and it is expressed as: $(\_^\star \cdot A \cdot \overline{B})^\star$, where $\_$ is a wildcard matching to any events; $\star$ denotes a repeated pattern; $\overline{B}$ denotes the negation of an effect

[1] A deep handler is persistent: after it has handled one effect, it remains installed, as the topmost frame of the captured continuation [10,11].

B. *"Function* `send(int n)` *terminates when* `n` *is non-negative, otherwise it does not terminate"* is expressed as: $n{\geq}0 \wedge (\_)^{\star} \vee n{<}0 \wedge (\_)^{\omega}$, which is beyond LTL.

Having *ContEffs* as the specification language, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal property Φ', does $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ hold[2]? In a typical verification context, checking the inclusion/entailment between the program effects $\Phi^{\mathcal{P}}$ and the valid traces Φ' proves that: the program $\mathcal{P}$ will never lead to unsafe traces which violate Φ'.

To effectively check $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$, we deploy a purely algebraic TRS inspired by Antimirov and Mosses' algorithm [12], which was originally designed for deciding the inequalities of regular expressions. Our TRS shows the ability to solve inclusions beyond the expressiveness of finite-state automata, also suggests that it is a better average-case algorithm than those based on automata theory.

We aim to lay the foundation for a practical verification system that is precise, concise, and modular to prove temporal properties of effectful programs. To the best of the authors' knowledge, this work is the first to provide an extensive temporal verification framework for programs with user-defined effects and handlers. We summarize our main contributions as follows:

1. **The Continuous Effect (*ContEffs*):** We define the syntax and semantics of *ContEffs*, to be the specification language, which captures the temporal behaviors of given higher-order programs with algebraic effects.
2. **Front-End Effects Inference:** Targeting a ML-like language with the presence of algebraic effects [4,13], we establish a set of forward rules, to compositionally infer the program's temporal behaviors. The inference process makes use of a fixpoint calculator and the back-end solver TRS.
3. **The Term Rewriting System (TRS):** To check the entailments (i.e., the language inclusion relation) between two *ContEffs*s, we present the rewriting rules, to prove the inferred effects against given temporal specifications.
4. **Implementation and Evaluation:** We prototype the proposed verification system based on the latest Multicore OCaml (4.12.0) implementation. We prove its correctness and present case studies investigating *ContEffs*' expressiveness and the potential for various extensions.

2 Overview

2.1 A Sense of *ContEffs* in File I/O

We define Hoare-triple style specifications, marked in `lavender`, for each program, which leads to a compositional verification strategy, where temporal reasoning can be done locally. We model an abstract form of file I/O in Fig. 2. Effects `Open` and `Close` are both declared to be *performed* with a value of type `int`, indexing the operated file.

[2] The inclusion notation $\sqsubseteq$ is formally defined in Definition 3.

Function `open_file` takes an argument `n`. Its precondition uses a wildcard '_' under a Kleene star, indicating that any finite number/kind of effects is allowed to have occurred *before* the call to `open_file`. In other words, it is always possible to open a file. Its postcondition indicates that it performs the effect `Open` applied with `n`.

The precondition of `close_file` states that it can only be called after such a history trace where the `n`th file has been requested to be `Open`ed, and not been requested to be `Close`ed[3].

We use `.` to denote the sequential composition of effect traces, `!` denotes the emission of a certain effect, and `~` denotes the negation of a certain effect label.

```
effect Open : int -> unit
effect Close: int -> unit

let open_file n
(*@ req _^* @*)
(*@ ens Open(n)! @*)
= perform (Open n)
let close_file n
(*@ req _^*.Open(n)!
         .(~Close(n)!)^* @*)
(*@ ens Close(n)! @*)
= perform (Close n)

let file_9 ()
(*@ req emp @*)
(*@ ens Open(9)!.Close(9)!@*)
= open_file 9;
  close_file 9
```

Fig. 2. A simple file I/O example.

The precondition of `file_9`: `emp`, stands for an empty trace, which means no history trace is *allowed* by the calling site of function `file_9`. We formalize this idea of *being allowed* as an entailment relation between specifications in Sec. 5. The verification fails when the real implementation violates the specifications.

2.2 Effects Inferences via a Fixpoint Calculation

We continue to examine a variant of the so-called "recursive cow" benchmark program [9] in Fig. 3., which generates an infinite trace. The handling of effects `Foo` and `Goo` are notable because their resumption carry closures back to the suspended points, which in turn perform effects when fully applied.

We argue informally that `loop` is non-terminating. This is because the invocation of `f_g ()` performs `Foo`, which obtains the resumed closure (defined in line 16) and stores it in the variable `f`. Then the application to `f` in turn performs `Goo`. The performing of `Goo` brings us to the handler at line

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

Fig. 3. Another Loop.

[3] `close_file`'s precondition prevents closing files that are not opened. The constraints can be strengthened or loosened as needed. For example, to prevent opening a file which is already opened, we need to strengthen `open_file`'s precondition accordingly.

18, which resumes a closure that performs `Foo` when applied. The resulting postcondition, deploys the ω operator, states that `loop` *finally* performs an infinite succession of alternating `Foo` and `Goo` effects. In fact, our fixpoint calculator computes the final effects for `loop` as $\mathtt{Foo} \cdot \mathtt{Goo} \cdot \mathtt{Goo} \cdot \mathtt{Foo} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega$, which entails the declared postcondition (c.f. Fig. 4.).

Loops like these between handler and callee are generally caused by performing effects in the recovery closure when handling an effect, that results in a cycle back to that same (deep) handler. However, resuming with a closure, is a useful pattern for inverting control between handler and callee, does give rise to this trap. Our fixpoint analysis and specifications are aimed at capturing such situations, which have not been extensively explored.

2.3 The TRS: To Prove Effects Inclusions

The rewriting system proposed by Antimirov and Mosses [14] decides inequalities of regular expressions (REs) through an iterated process of checking the inequalities of their *partial derivatives* [15]. There are two basic rules: `[DISPROVE ]`, which infers false from trivially inconsistent inequalities; and `[UNFOLD]`, which applies Definition 1 to generate new inequalities. In detail, given Σ is the whole set of the alphabet, $D_A(r)$ is the partial derivative of r w.r.t the event A.

Definition 1 (REs Inequality). *For REs r, s, $r \preceq s \Leftrightarrow \forall(A \in \Sigma).\ D_A(r) \preceq D_A(s)$.*

Similarly, we formally define the inclusion of *ContEffs* in Definition 3.

Next we present the effects inclusion, generated from Fig. 3, proving process for the post condition checking in Fig. 4. Termination is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization*. We use $\spadesuit$ to indicate such pairings. The rewriting rules are defined in Sec. 5. In particular, the rule [*Reoccur*] finds the syntactic identity from the internal proof tree, for the current open goal [16].

$$\begin{array}{cl}
\mathtt{Foo} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega \sqsubseteq \_^\star \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega\,\spadesuit \vee (\mathtt{Foo} \cdot \mathtt{Goo})^\omega & \\
\hdashline
 & \mathit{fst}{=}\mathit{Goo} \\
\underline{\mathtt{Goo}} \cdot \mathtt{Foo} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega \sqsubseteq \_^\star \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega \vee \mathtt{Goo} \cdot (\mathtt{Foo} \cdot \mathtt{Goo})^\omega & \\
\hdashline
 & \mathit{fst}{=}\mathit{Foo} \\
\underline{\mathtt{Foo}} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega \sqsubseteq \_^\star \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega\,\spadesuit & \\
\hdashline
 & \mathit{fst}{=}\mathit{Goo} \\
\underline{\mathtt{Goo}} \cdot \mathtt{Foo} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega \sqsubseteq \_^\star \cdot (\mathtt{Foo} \cdot \mathtt{Goo})^\omega \vee (\mathtt{Foo} \cdot \mathtt{Goo})^\omega & \\
\hdashline
 & \mathit{fst}{=}\mathit{Goo} \\
\underline{\mathtt{Goo}} \cdot \mathtt{Goo} \cdot \mathtt{Foo} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega \sqsubseteq \_^\star \cdot (\mathtt{Foo} \cdot \mathtt{Goo})^\omega \vee \mathtt{Goo} \cdot (\mathtt{Foo} \cdot \mathtt{Goo})^\omega & \\
\hdashline
 & \mathit{fst}{=}\mathit{Foo} \\
\underline{\mathtt{Foo}} \cdot \mathtt{Goo} \cdot \mathtt{Goo} \cdot \mathtt{Foo} \cdot (\mathtt{Goo} \cdot \mathtt{Foo})^\omega \sqsubseteq \_^\star \cdot (\mathtt{Foo} \cdot \mathtt{Goo})^\omega &
\end{array}$$

Fig. 4. Proving the postcondition of `loop ()`.

3 Language and Specifications

3.1 The Target Language

Syntax. We target a minimal, ML-like (typed, higher-order, call-by-value) core pure language, defined in Fig. 5. Here, c, x and A are meta-variables ranging respectively over integer constants, variables, and labels of effects.

A program $\mathcal{P}$ comprises a list of effect declarations eff^* and a list of method definitions meth^*; the $*$ superscript denotes a finite, possibly empty list of items. Programs are typed according to basic types τ. Each method *meth* has a name *mn*, an expression body e, and pre- and postconditions Φ_{pre} and Φ_{post} (the syntax of effect specifications Φ is given in Fig. 7). Constructs like sequencing are defined via elaboration to more primitive forms.

$$
\begin{array}{lrl}
(\mathit{Program}) & \mathcal{P} ::= & \mathit{eff}^* \ \mathit{meth}^* \\
(\mathit{Effect\ Declarations}) & \mathit{eff} ::= & A : \tau \\
(\mathit{Method\ Definition}) & \mathit{meth} ::= & \tau \ mn \ (\tau \ v) \ [\mathbf{req} \ \Phi_{pre} \ \mathbf{ens} \ \Phi_{post}] \ \{e\} \\
(\mathit{Types}) & \tau ::= & \mathit{bool} \mid \mathit{int} \mid \mathit{unit} \mid \tau_1 \rightarrow \tau_2 \\
(\mathit{Values}) & v ::= & c \mid x \mid \lambda x \Rightarrow e \\
(\mathit{Handler}) & h ::= & (\mathit{return} \ x \mapsto e \ \mid \ \mathit{ocs}) \\
(\mathit{Operation\ Cases}) & \mathit{ocs} ::= & \emptyset \mid \{\mathit{effect} \ A(x, \kappa) \mapsto e\} \uplus \mathit{ocs} \\
(\mathit{Expressions}) & e ::= & v \mid v_1 \ v_2 \mid \mathit{let} \ x{=}v \ \mathit{in} \ e \mid \mathit{if} \ v \ \mathit{then} \ e_1 \ \mathit{else} \ e_2 \mid \\
 & & \mathit{perform} \ A(v, \lambda x \Rightarrow e) \mid \mathit{match} \ e \ \mathit{with} \ h \mid \mathit{resume} \ v
\end{array}
$$

$(\mathit{Selected\ Elaborations})$

$$
\begin{array}{rcl}
e_1; e_2 & \Longrightarrow & \mathit{let} \ (){=}e_1 \ \mathit{in} \ e_2 \\
e_1 \ e_2 & \Longrightarrow & \mathit{let} \ f{=}e_1 \ \mathit{in} \ \mathit{let} \ x{=}e_2 \ \mathit{in} \ (f \ x) \\
\mathit{perform} \ A(e_1, \lambda y \Rightarrow e_2) & \Longrightarrow & \mathit{let} \ x{=}e_1 \ \mathit{in} \ \mathit{perform} \ A(x, \lambda y \Rightarrow e_2) \\
\mathit{let} \ x{=}\mathit{perform} \ A(v, \lambda y \Rightarrow e_1) \ \mathit{in} \ e_2 & \Longrightarrow & \mathit{perform} \ A(v, \lambda y \Rightarrow \mathit{let} \ x{=}e_1 \ \mathit{in} \ e_2) \\
\mathit{perform} \ A(v) & \Longrightarrow & \mathit{perform} \ A(v, \lambda x \Rightarrow x)
\end{array}
$$

$$c \in \mathbb{Z} \cup \mathbb{B} \cup \mathbf{unit} \qquad x, y, mn, \kappa \in \mathbf{var} \qquad A \in \Sigma$$

Fig. 5. Syntax of expressions.

$$(\mathit{Evaluation\ contexts}) \quad E ::= \square \mid \mathit{let} \ x{=}E \ \mathit{in} \ e \mid \mathit{match} \ E \ \mathit{with} \ h$$

$$
\begin{array}{lrcl}
(\mathit{Reduction\ rules}) & E[e_1] & \longrightarrow & E[e_2] \text{ if } e_1 \longrightarrow e_2 \\
 & \mathit{let} \ x{=}v \ \mathit{in} \ e & \longrightarrow & e[v/x] \\
 & (\lambda x \Rightarrow e) \ v & \longrightarrow & e[v/x] \\
 & \mathit{if} \ \mathit{true} \ \mathit{then} \ e_1 \ \mathit{else} \ e_2 & \longrightarrow & e_1 \\
 & \mathit{if} \ \mathit{false} \ \mathit{then} \ e_1 \ \mathit{else} \ e_2 & \longrightarrow & e_2 \\
 & \mathit{match} \ v \ \mathit{with} \ h & \longrightarrow & e[v/x] \ \ \mathit{if} \ (\mathit{return} \ x \mapsto e) \in h \\
 & \mathit{match} \ (\mathit{perform} \ A(v, \lambda y \Rightarrow e_1)) \ \mathit{with} \ h & \longrightarrow & e_2[v/x][(\lambda y \Rightarrow \mathit{match} \ e_1 \ \mathit{with} \ h)/\kappa] \\
 & & & \mathit{if} \ (\mathit{effect} \ A(x, \kappa) \mapsto e_2) \in h \\
 & \mathit{match} \ (\mathit{perform} \ A(v, \lambda y \Rightarrow e_1)) \ \mathit{with} \ h & \longrightarrow & \mathit{perform} \ A(v, \lambda y \Rightarrow \mathit{match} \ e_1 \ \mathit{with} \ h) \\
 & & & \mathit{if} \ A \notin h
\end{array}
$$

Fig. 6. Evaluation contexts and reduction rules

Operational Semantics. The reduction rules up to those for *match* are standard. Matching on a pure value results in the body of the always-present *return* handler being executed, with x bound to the value. The next two cases define how effects are performed and handled, but before covering them, we first explain how the expression *perform* $A(v, \lambda x \Rightarrow e)$ works informally: it performs the effect A (e.g. a shared-memory read) with argument v (e.g. the memory location to be read). The result value of the effect (e.g. the contents of the memory location) is then bound to x and evaluation resumes with the continuation e. Note that how exactly the read is *implemented* is defined by handlers which enclose the *perform*.

With that in mind, there are two cases when matching on an effectful expression. If the effect A is handled by an appropriate case in an enclosing handler, both value and continuation are substituted into the body of the case – note that the continuation contains an identical handler (making the enclosing handler *deep*). Otherwise, if the effect is unhandled, reduction proceeds with the current *match* "pushed" into the continuation, to handle subsequent *performs*.

3.2 The Specification Language

Syntax. We enrich a Hoare-style verification system with effect specifications, using the notation $\{\texttt{req}\ \Phi_{pre}\ \texttt{ens}\ \Phi_{post}\}$ for function pre- and postconditions. As defined in Fig. 7, Φ is a set of disjunctive tuples including a pure formula π, an event sequence θ, and a return value v.

$$
\begin{array}{rrcl}
(\mathit{ContEffs}) & \Phi & ::= & \bigvee(\pi, \theta, v) \\
(\mathit{Parameterized\ Label}) & l & ::= & \Sigma(v) \\
(\mathit{Event\ Sequences}) & \theta & ::= & \bot \mid \epsilon \mid ev \mid Q \mid \theta_1 \cdot \theta_2 \mid \theta_1 \vee \theta_2 \mid \theta^\star \mid \theta^\infty \mid \theta^\omega \\
(\mathit{Single\ Events}) & ev & ::= & \_ \mid l \mid \overline{l} \\
(\mathit{Placeholders}) & Q & ::= & l! \mid l?(v) \\
(\mathit{Pure\ formulae}) & \pi & ::= & \mathit{True} \mid \mathit{False} \mid R(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \pi_1 \Rightarrow \pi_2 \\
(\mathit{Terms}) & t & ::= & n \mid x \mid t_1 + t_2 \mid t_1 \text{-} t_2
\end{array}
$$

$x \in \textbf{var}$ $\quad$ $(\mathit{Finite\ Kleene\ Star})\ \star$ $\quad$ $(\mathit{Finite/Infinite})\ \infty$ $\quad$ $(\mathit{Infinite})\ \omega$

Fig. 7. Syntax of *ContEffs*.

A is an effect label drawn from Σ, a finite set of user-defined effect labels. A *parameterized label* is an effect label together with a value argument v. An *event* ev is an assertion about the (non-)occurrence of an individual, *handled* effect.

Placeholders Q stand for *traces* (sequences of events). The two kinds of placeholders are *unhandled* effects $l!$, which may give rise to further effects upon being handled, and $l?(v)$, which describes the trace that results when l is resumed with a higher-order function, and this function is applied to v. Placeholders enable modular verification, allowing higher-order *perform* sites to be described independently of any particular handler. They are only instantiated while verifying handlers, using the fixed-point reasoning (Sect. 4.2).

Effect sequences θ can be constructed by *false* ($\bot$); the empty trace ϵ; a single event ev; a placeholder Q; a sequence concatenation $\theta_1 \cdot \theta_2$; and sequence disjunction $\theta_1 \vee \theta_2$. Effect sequences can be also constructed by $\star$, representing finite (zero or more) repetition of a trace; by ω, representing an infinite repetition of a trace; or by ∞, representing an overapproximation of both finite and infinite possibilities [17]. Although $\theta^\star$ and θ^ω are subsumed by θ^∞, integrating all of the operators makes the specification language more flexible and precise. It also makes the logic conveniently fuse traditional linear temporal logics.

Pure formulae π are Presburger arithmetic formulae. $R(t_1, t_2)$ is a binary relation ($R \in \{=, >, <, \geq, \leq\}$). Terms are constant integer values n, integer variables x, and additions and subtractions of terms.

Semantic Model of Effect Sequences. To define the model, *var* is the set of program variables, *val* is the set of primitive values, α is the set of concrete events drawn from single events l or placeholders Q. Let $\mathcal{E}, \varphi \models \Phi$ denote the *models* relation, i.e., the context $\mathcal{E}$ and linear temporal events φ satisfy the effect specification Φ, where $\mathcal{E}$ records the stack status and the bindings from variables to placeholders, $\mathcal{E} \triangleq var \rightarrow (val \cup Q)$; and φ is a list of events, $\varphi \triangleq [\alpha]$.

Since the return value in effect specifications is irrelevant to the semantic model, we define $\mathcal{E}, \varphi \models (\pi, \theta)$ to be $\mathcal{E}, \varphi \models (\pi, \theta, v)$ for some return value v.

The semantics of effect sequences is defined in Fig. 8. [] is an empty sequence; $[l]$ is the sequence that contains one parameterized label l; ++ is the append operation of two effect sequences; and $\bigvee j$ is a disjunction of parameterized labels j. Comparisons between labels use simple lexical equivalence.

$$
\begin{array}{lll}
\mathcal{E}, \varphi \models \Phi & \textit{iff} & \exists (\pi, \theta, v) \in \Phi.\ \mathcal{E}, \varphi \models (\pi, \theta, v) \\
\mathcal{E}, \varphi \models (\pi, \epsilon) & \textit{iff} & [\![\pi]\!]_{\mathcal{E}} = \textit{True and } \varphi = [] \\
\mathcal{E}, \varphi \models (\pi, \_) & \textit{iff} & [\![\pi]\!]_{\mathcal{E}} = \textit{True and } \exists l \in \Sigma(v),\ \varphi = [l] \\
\mathcal{E}, \varphi \models (\pi, l) & \textit{iff} & [\![\pi]\!]_{\mathcal{E}} = \textit{True and } \varphi = [l] \\
\mathcal{E}, \varphi \models (\pi, \overline{l}) & \textit{iff} & [\![\pi]\!]_{\mathcal{E}} = \textit{True and } \mathcal{E}, \varphi \models \bigvee j \textit{ where } j \in \Sigma(v) \textit{ and } j \neq l \\
\mathcal{E}, \varphi \models (\pi, Q) & \textit{iff} & [\![\pi]\!]_{\mathcal{E}} = \textit{True and } \varphi = [Q] \\
\mathcal{E}, \varphi \models (\pi, \theta_1 \cdot \theta_2) & \textit{iff} & \exists \varphi_1, \varphi_2.\ \varphi = \varphi_1 \texttt{++} \varphi_2 \textit{ and } \mathcal{E}, \varphi_1 \models (\pi, \theta_1) \textit{ and } \mathcal{E}, \varphi_2 \models (\pi, \theta_2) \\
\mathcal{E}, \varphi \models (\pi, \theta_1 \vee \theta_2) & \textit{iff} & \mathcal{E}, \varphi \models (\pi, \theta_1) \textit{ or } \mathcal{E}, \varphi \models (\pi, \theta_2) \\
\mathcal{E}, \varphi \models (\pi, \theta^\star) & \textit{iff} & \mathcal{E}, \varphi \models (\pi, \epsilon) \textit{ or } \mathcal{E}, \varphi \models (\pi, \theta \cdot \theta^\star) \\
\mathcal{E}, \varphi \models (\pi, \theta^\infty) & \textit{iff} & \mathcal{E}, \varphi \models (\pi, \theta^\star) \textit{ or } \mathcal{E}, \varphi \models (\pi, \theta^\omega) \\
\mathcal{E}, \varphi \models (\pi, \theta^\omega) & \textit{iff} & \mathcal{E}, \varphi \models (\pi, \theta \cdot \theta^\omega) \\
\mathcal{E}, \varphi \models (\textit{False}, \bot) & \textit{iff} & \textit{false}
\end{array}
$$

Fig. 8. Semantics of effect sequences.

3.3 Instrumented Semantics

To facilitate the soundness proof in Theorem 1 for the verification rules presented in Sect. 4, we also define an instrumented reduction relation $\xrightarrow{i}$, which operates on program states of the form $\lceil e, \mathcal{E}, \varphi \rceil$, where an expression is associated with a context and the trace of effects performed in the course of its execution. $\xrightarrow{i}{}^*$ denotes its reflexive, transitive closure. Here, given $e \longrightarrow e'$ and a most general high-order effects signature $(A : \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)) \in \mathcal{P}$:

$$\frac{e = v_1\ v_2 \qquad \mathcal{E}(v_1){=}A(v)?}{\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e', \mathcal{E}, \varphi{+}{+}[A(v)?(v_2)] \rceil}[\mathit{Inst\text{-}App}] \quad \frac{e = \mathit{let}\ x{=}v\ \mathit{in}\ e_1}{\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e', (x{\mapsto}v){::}\mathcal{E}, \varphi \rceil}[\mathit{Inst\text{-}Bind}]$$

$$\frac{e = \mathit{match\ perform}\ A(v, \lambda x{\Rightarrow}e_1)\ \mathit{with}\ h \qquad A \notin h}{\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e', (x{\mapsto}A(v)?){::}\mathcal{E}, \varphi{+}{+}[A(v)!] \rceil}\ [\mathit{Inst\text{-}Escape}]$$

$$\frac{e = \mathit{match\ perform}\ A(v, \lambda x{\Rightarrow}e_1)\ \mathit{with}\ h \qquad A \in h}{\lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i} \lceil e', \mathcal{E}, \varphi{+}{+}[A(v)] \rceil}\ [\mathit{Inst\text{-}Caught}]$$

4 Forward Verification

An overview of our automated verification system is given in Fig. 9. It consists of a Hoare-style forward verifier and a TRS. The input of the forward verifier is a target program annotated with temporal specifications written in *ContEffs*.

The input of the TRS is a pair of effects LHS and RHS, referring to the inclusion LHS $\sqsubseteq$ RHS to be checked *(LHS for left-hand-side trace, and RHS for right-hand-side trace)*. The verifier calls the TRS to prove produced inclusions.

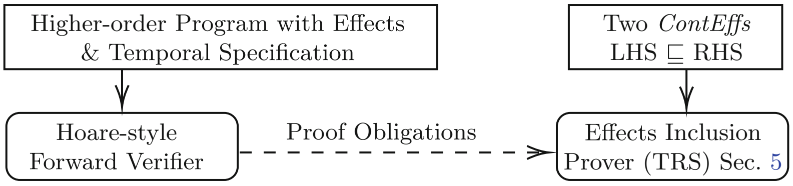
Higher-order Program with Effects & Temporal Specification
Two *ContEffs* LHS ⊑ RHS
Hoare-style Forward Verifier
Proof Obligations
Effects Inclusion Prover (TRS) Sec. 5

Fig. 9. System overview. Rounded boxes are the main procedures. Rectangular boxes describe the inputs to the procedures. The verification relies on the TRS (dash line).

We formalize a set of syntax-directed forward verification rules for the core language. $\mathcal{P}$ denotes the program being checked. With pre/postconditions declared for each method in $\mathcal{P}$, we apply modular verification to a method's body using Hoare-style triples $\mathcal{E} \vdash \{\Phi\}\ e\ \{\Phi'\}$ where $\mathcal{E}$ is the context; if Φ describes the effects which have been performed since the beginning of $\mathcal{P}$, if e terminates, Φ' describes the effects that will have been performed after.

4.1 Forward Verification Rules

In [*FV-Meth*], the rule computes the final effects Φ from the method body, and checks the inclusion between Φ and the declared specifications. Note that for succinctness, the user-provided Φ_{post} only denotes the *extension* of the effects from executing the method body. Formally, $\mathcal{E} \vdash \{\Phi_{pre}\}\ e\ \{\Phi_{pre} \cdot \Phi_{post}\}$ is a valid triple.

$$\frac{\mathcal{E} \vdash \{\Phi_{pre}\}\ e\ \{\Phi\} \qquad \Phi \sqsubseteq \Phi_{pre} \cdot \Phi_{post}}{\vdash \tau\ mn\ (\tau\ v)\ [\mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}]\ \{e\}}\ [\mathit{FV\text{-}Meth}]$$

Definition 2 (*ContEffs* Concatenation). *Given two* ContEffs Φ_1 *and* Φ_2, $\Phi_1 \cdot \Phi_2 = \{(\pi_1 \wedge \pi_2, \theta_1 \cdot \theta_2, v_2) \mid (\pi_1, \theta_1, v_1) \in \Phi_1, (\pi_2, \theta_2, v_2) \in \Phi_2\}$

[*FV-Perform*] concatenates a placeholder to the current effects, where $\Phi \cdot A(v)! \equiv \{(\pi, \theta \cdot A(v)!, v) \mid (\pi, \theta, v) \in \Phi\}$, then extends the environment by binding x to $A(v)?$, referring to the resumed value of performing $A(v)$.

$$\frac{\Phi' {=} \Phi \cdot A(v)! \qquad (x {\mapsto} A(v)?){::}\mathcal{E} \vdash \{\Phi'\}\ e\ \{\Phi''\}}{\mathcal{E} \vdash \{\Phi\}\ \mathit{perform}\ A(v, \lambda x {\Rightarrow} e)\ \{\Phi''\}}\ [\mathit{FV\text{-}Perform}]$$

For applications $v_1\, v_2$, if v_1 is a function definition with annotated specifications, [*FV-Call*] checks whether the instantiated precondition of callee, $\Phi_{pre}[v_2/v]$, is satisfied by the current effects state, then it obtains the next effects state by concatenating the instantiated postcondition, $\Phi_{post}[v_2/v]$, to the current effects state; if v_1 maps to $l?$, [*FV-App*] concatenates $l?(v_2)$ into the current effect state, referring to the effects generated by applying v_2 to the value resumed from performing l. [*FV-Value*] updates the current return value.

$$\frac{\mathcal{E}(v_1) {=} \tau\ mn\ (\tau\ v)\ [\mathbf{req}\ \Phi_{pre}\ \mathbf{ens}\ \Phi_{post}]\ \{e\} \qquad \Phi \sqsubseteq \Phi_{pre}[v_2/v]}{\mathcal{E} \vdash \{\Phi\}\ v_1 v_2\ \{\Phi \cdot \Phi_{post}[v_2/v]\}}\ [\mathit{FV\text{-}Call}]$$

$$\frac{\mathcal{E}(v_1) {=} l? \qquad \theta' {=} l?(v_2)}{\mathcal{E} \vdash \{\Phi\}\ v_1 v_2\ \{\Phi \cdot \theta'\}}\ [\mathit{FV\text{-}App}] \qquad \frac{\Phi' = \{(\pi, \theta, v') \mid (\pi, \theta, v) \in \Phi\}}{\mathcal{E} \vdash \{\Phi\}\ v'\ \{\Phi'\}}\ [\mathit{FV\text{-}Value}]$$

[*FV-If-Else*] unions the effects from both branches, where $\Phi \wedge \pi' \equiv \{(\pi \wedge \pi', \theta, v) \mid (\pi, \theta, v) \in \Phi\}$. [*FV-Let*] extends $\mathcal{E}$ with x binding to v.

$$\frac{\mathcal{E} \vdash \{\Phi {\wedge} (v {=} \mathit{true})\}\ e_1\ \{\Phi_1\} \qquad \mathcal{E} \vdash \{\Phi {\wedge} (v {=} \mathit{false})\}\ e_2\ \{\Phi_2\}}{\mathcal{E} \vdash \{\Phi\}\ \mathit{if}\ v\ \mathit{then}\ e_1\ \mathit{else}\ e_2\ \{\Phi_1\} \cup \{\Phi_2\}}\ [\mathit{FV\text{-}If\text{-}Else}]$$

$$\frac{(x {\mapsto} v){::}\mathcal{E} \vdash \{\Phi\}\ e\ \{\Phi'\}}{\mathcal{E} \vdash \{\Phi\}\ \mathit{let}\ x{=}v\ \mathit{in}\ e\ \{\Phi'\}}\ [\mathit{FV\text{-}Let}]$$

[*FV-Match*] computes the effects of e using the initial state $\{(\mathit{True}, \epsilon, ())\}$, then deploys the fixpoint algorithm to compute the final effects after been handled by h. The notion $\heartsuit$ is a special event marking the end of the traces, which is essential when distinguishing the zero/one/multi-shots continuations.

$$\frac{\mathcal{E} \vdash \{(\mathit{True}, \epsilon, ())\}\ e\ \{\Phi'\} \qquad \Phi''=\{(\pi, \theta \cdot \heartsuit, v) \mid (\pi, \theta, v) \in \Phi'\} \quad \mathcal{E}, h \vdash_{\mathit{fix}} \Phi'' \rightsquigarrow \Phi_{\mathit{fix}}\ \text{(cf. Sec. 4.2)}}{\mathcal{E} \vdash \{\Phi\}\ \mathit{match}\ e\ \mathit{with}\ h\ \{\Phi \cdot \Phi_{\mathit{fix}}\}}\ [\mathit{FV\text{-}Match}]$$

4.2 Fixpoint Computation

Given any effect Φ and fixed environment $\mathcal{E}$ and handler $\mathcal{H}$, the relation $\mathcal{E}, \mathcal{H} \vdash_{\mathit{fix}} \Phi \rightsquigarrow \Phi_{\mathit{fix}}$ concludes the fixpoint effects Φ_{fix} via the following rule, where

$$\frac{\forall(\pi, \theta, v) \in \Phi.\ \|\mathcal{E}, \epsilon, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \theta, v) \rightsquigarrow \Phi'}{\mathcal{E}, \mathcal{H} \vdash_{\mathit{fix}} \Phi \rightsquigarrow \bigcup \Phi'}\ [\mathit{Fix\text{-}Disj}]$$

for all execution tuples (π, θ, v) from Φ, given $\mathcal{E}$ and $\mathcal{H}$, it is reduced to Φ'. Their relation is captured by: $\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \theta, v) \rightsquigarrow \Phi'$, where θ_{his} is the history trace and initialized by ϵ. The final result Φ_{fix} is a union set of all the Φ'.

Rule [*Fix-Normal*] is applied when the trace is reduced to the ending mark $\heartsuit$, which indicates that the execution of the handled program is finished. In this case, the resulting state Φ' is achieved by computing the strongest post condition of $e_{ret}[v/x]$ from the starting state $\{(\pi, \theta_{his}, v)\}$.

$$\frac{(\mathit{return}\ x \mapsto e_{ret}) \in \mathcal{H} \qquad ([x{\mapsto}v])::\mathcal{E} \vdash \{(\pi, \theta_{his}, v)\}\ e_{ret}\ \{\Phi'\}}{\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \heartsuit, v) \rightsquigarrow \Phi'}\ [\mathit{Fix\text{-}Normal}]$$

Rule [*Fix-Unfold-Skip*] is applied when the starting events α are handled effects ev, or placeholders corresponding to the effects cannot be handled by the current handler. In this case, the rule simple achieves α into the history context θ_{his} and continues to reason about the tail of the trace, i.e., θ.

$$\frac{\alpha \in \{ev, l!, l?(v')\}\ (l \notin \mathcal{H}) \qquad \|\mathcal{E}, \theta_{his} \cdot \alpha, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \theta, v) \rightsquigarrow \Phi'}{\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \alpha \cdot \theta, v) \rightsquigarrow \Phi'}\ [\mathit{Fix\text{-}Unfold\text{-}Skip}]$$

Rule [*Fix-Unfold-Handle*] is applied when the starting events α are unhandled effects $l!$ which can be handled by the current handler. In this case, the rule uses the relation $\mathcal{E}', \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ e\ \langle\Phi'\rangle$ to reason about the handling code e, where $\mathcal{E}'$ extends $\mathcal{E}$ with $x{\mapsto}v$. Note that, here the rule achieves l into the history context, indicating that the emission $l!$ is handled.

$$\frac{\begin{array}{ccc}\alpha \in \{l!\} & (\mathit{effect}\ A(x,\kappa) \mapsto e) \in \mathcal{H} & (l{=}A(v')) \\ \mathcal{E}'{=}(x{\mapsto}v){::}\mathcal{E} & \multicolumn{2}{l}{\mathcal{E}', \mathcal{H}, \theta \vdash_h \langle(\pi, \theta_{his} \cdot l, v)\rangle\ e\ \langle\Phi'\rangle \text{ (cf. Sec. 4.3)}}\end{array}}{\|\mathcal{E}, \theta_{his}, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \alpha \cdot \theta, v) \rightsquigarrow \Phi'}\ [\mathit{Fix\text{-}Unfold\text{-}Handle}]$$

4.3 Reasoning in the Handling Program

Rules for $\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ e\ \langle\Phi'\rangle$ (where $\mathcal{D}$ stands for the not-yet-handled continuation, of the type θ) are mostly similar to the top-level forward relation $\mathcal{E} \vdash \{\Phi\}\ e\ \{\Phi'\}$, except for the rules:

$$\frac{\begin{array}{cc}\forall(\pi,\theta,v) \in \Phi & \|\mathcal{E}, \theta, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \mathcal{D}[v'/l?], v) \rightsquigarrow \Phi' \\ \multicolumn{2}{c}{\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi'\rangle\ e\ \langle\Phi''\rangle}\end{array}}{\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ \mathit{let}\ x{=}\kappa\ v'\ \mathit{in}\ e\ \langle\Phi''\rangle}\ [\mathit{Handle\text{-}Resume}]$$

$$\frac{\Phi'{=}\{(\pi,\theta,v') \mid (\pi,\theta,v) \in \Phi\}}{\mathcal{E}, \mathcal{H}, \mathcal{D} \vdash_h \langle\Phi\rangle\ v'\ \langle\Phi'\rangle}\ [\mathit{Handle\text{-}Value}]$$

In [*Handle-Resume*], all the placeholders $l?$ shown in the continuation $\mathcal{D}$ can be finally instantiated by κ's argument value, v'. Possible loops are also captured in this step, when $\mathcal{D}[v'/l?]$ produces the effects' emissions which has already been handled. The final result Φ'' is achieved by reasoning e after handling the rest continuation. Note that if the handling program directly returns a single value, the rule [*Handle-Value*] abandons the continuation $\mathcal{D}$ completely, which is intuitively why we are able to handle exceptions (zero-shot continuations). The rest of the rules and a demonstration example are presented in Appendix A.

Lemma 1 (Soundness of the Fixpoint Computation). *Given an effect Φ, with the environment $\mathcal{E}$ and handler $\mathcal{H}$. Φ_{fix} is the updated version of Φ, where all Φ's placeholders – which can be handled by $\mathcal{H}$ – are handled as $\mathcal{H}$ defines.*

Formally, $\forall\mathcal{E}, \forall\mathcal{H}, \forall\Phi$, *if* $\mathcal{E}, \mathcal{H} \vdash_{\mathit{fix}} \Phi \rightsquigarrow \Phi_{\mathit{fix}}$ *is valid, then:*

when Φ *is a set,* $\Phi_{\mathit{fix}}{=}\{\|\mathcal{E}, \epsilon, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi,\theta,v) \rightsquigarrow \Phi' \mid (\pi,\theta,v) \in \Phi\}$; (1)

when $\Phi{=}(\pi,\theta,v), \alpha{=}\mathit{fst}(\theta), \theta_{his}$ *is the handled trace,*

if $\alpha{=}\heartsuit : ([x{\mapsto}v]){::}\mathcal{E}{\vdash}\{(\pi, \theta_{his}, v)\}e_{ret}\{\Phi'\}$ *is valid, given* $(\mathit{return}\ x{\mapsto}e_{ret}){\in}\mathcal{H}$; (2)

if $\alpha{\in}\{ev, l!, l?(v')\}\ (l{\notin}\mathcal{H}) :\ \|\mathcal{E}, \theta_{his} \cdot \alpha, \mathcal{H}\| \vdash_{\mathit{fix}} (\pi, \mathcal{D}_\alpha(\theta), v) \rightsquigarrow \Phi'$ *is valid;* (3)

if $\alpha{\in}\{l!\}\ (l{\in}\mathcal{H}) :\ (x{\mapsto}v){::}\mathcal{E}, \mathcal{H}, \mathcal{D}_\alpha(\theta) \vdash_h \langle(\pi, \theta_{his} \cdot l, v)\rangle\ e\ \langle\Phi'\rangle$ *is valid, given* $(\mathit{effect}\ A(x,\kappa) \mapsto e){\in}\mathcal{H}$. (4)

Proof. See Appendix B.

Theorem 1 (Soundness of Verification Rules). *Given an expression e, the linear effect trace produced by the real execution of e satisfies the effect specification derived via the forward verification rules.*

$$\textit{Formally}, \forall e, \forall \mathcal{E}, \forall \varphi, \forall \Phi \textit{ given } \lceil e, \mathcal{E}, \varphi \rceil \xrightarrow{i}{}^{*} \lceil v, \mathcal{E}', \varphi' \rceil \textit{ and } \mathcal{E} \vdash \{\Phi\}\ e\ \{\Phi'\},$$
$$\textit{if } \mathcal{E}, \varphi \models \Phi \textit{ then } \mathcal{E}', \varphi' \models \Phi'.$$

Proof. See Appendix B.

5 Temporal Verification via a TRS

A TRS checks inclusions among logical terms, via an iterated process of checking the inclusions of their *partial derivatives* [15]. It is triggered i) prior to function calls for the precondition checking; and ii) at the end of verifying a function for postcondition checking. Given two effects Φ_1 and Φ_2, the TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid. During the rewriting process, the inclusions are of the form $\Omega \vdash \Phi_1 \sqsubseteq^{\theta} \Phi_2$, a shorthand for: $\Omega \vdash \theta \cdot \Phi_1 \sqsubseteq \theta \cdot \Phi_2$. To prove such inclusions amounts to checking whether all the possible traces in the antecedent Φ_1 are legitimately allowed in the possible traces from the consequent Φ_2. Ω is the proof context, i.e., a set of effect inclusion hypotheses, and θ is the history of effects from the antecedent that have been used to match the effects from the consequent. The inclusion checking is initially invoked with $\Omega{=}\{\}$ and $\theta{=}\epsilon$.

Effect Disjunction. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent. An inclusion with a disjunctive consequent succeeds if the antecedent entails any of the disjunctions. Note that the event sequences' entailment checking is irrelevant to the returning values.

$$\frac{\Omega \vdash (\pi, \theta) \sqsubseteq \Phi' \textit{ and } \ \Omega \vdash \Phi \sqsubseteq \Phi'}{\Omega \vdash (\pi, \theta, v) :: \Phi \sqsubseteq \Phi'}\ [\textit{LHS-OR}] \qquad \frac{\Omega \vdash (\pi, \theta) \sqsubseteq (\pi', \theta') \textit{ or } (\pi, \theta) \sqsubseteq \Phi'}{\Omega \vdash (\pi, \theta) \sqsubseteq (\pi', \theta', v') :: \Phi'}\ [\textit{RHS-OR}]$$

Definition 3 (*ContEffs* Inclusion). *For effects* (π_1, θ_1) *and* (π_2, θ_2), $(\pi_1, \theta_1) \sqsubseteq (\pi_2, \theta_2) \ \Leftrightarrow\ \pi_1 \Rightarrow \pi_2$ *and* $(\forall \alpha \in \Sigma).\ \mathcal{D}_\alpha(\theta_1) \sqsubseteq \mathcal{D}_\alpha(\theta_2)$.

Next we provide the definitions and implementations of auxiliary functions[4] *Nullable*(δ), *Infinitable*($\varkappa$), *First*(*fst*) and *Derivative*($\mathcal{D}$) respectively. Intuitively, the Nullable function $\delta(\Phi)$ returns a boolean value indicating whether θ contains the empty trace; the Infinitable function $\varkappa(\theta)$ returns a boolean value indicating whether θ is possibly infinite; the First function $\mathit{fst}(\theta)$ computes possible initial elements of θ; and the Derivative function $\mathcal{D}_\alpha(\theta)$ eliminates an event α[5] from the head of θ and returns what remains.

[4] The definitions are extended from [15], to be able to deal with placeholders and infinite traces, proposed in this work.

[5] α could be a single label l, a negated label $\overline{l}$, a wildcard $\_$, or a placeholder Q.

Definition 4 (Nullable). *Given any sequence θ, we recursively define $\delta(\theta)$*[6]

$$\delta(\epsilon)=\delta(\theta^{\star})=\delta(\theta^{\infty})=\mathit{true} \quad \delta(\theta_1 \cdot \theta_2)=\delta(\theta_1)\wedge\delta(\theta_2) \quad \delta(\theta_1 \vee \theta_2)=\delta(\theta_1)\vee\delta(\theta_2)$$

Definition 5 (Infinitable). *Given any sequence θ, we recursively define $\varkappa(\theta)$*[7]

$$\varkappa(\theta^{\infty})=\varkappa(\theta^{\omega})=\mathit{true} \quad \varkappa(\theta_1 \cdot \theta_2)=\varkappa(\theta_1)\vee\varkappa(\theta_2) \quad \varkappa(\theta_1 \vee \theta_2)=\varkappa(\theta_1)\vee\varkappa(\theta_2)$$

Definition 6 (First). *Let $fst(\theta)$ be the set of initial elements derivable from sequence represents all the traces contained in θ.*

$$fst(\bot)=fst(\epsilon)=\{\} \quad fst(ev)=\{ev\} \quad fst(Q)=\{Q\} \quad fst(\theta_1 \vee \theta_2)=fst(es_1)\cup fst(es_2)$$

$$fst(\theta_1 \cdot \theta_2)=\begin{cases} fst(es_1)\cup fst(es_2) & \mathit{if}\ \ \delta(\theta_1)=\mathit{true} \\ fst(\theta_1) & \mathit{if}\ \ \delta(\theta_1)=\mathit{false} \end{cases} \qquad fst(\theta^{\star})=fst(\theta^{\infty})=fst(\theta^{\omega})=fst(\theta)$$

Definition 7 (Partial Derivative). *The partial derivative $\mathcal{D}_\alpha(\theta)$ of effects θ w.r.t. an element α computes the effects for the left quotient, $\alpha^{-1}[\![\theta]\!]$*[8]*.*

$$\mathcal{D}_\alpha(\bot)=\bot \quad \mathcal{D}_\alpha(\epsilon)=\bot \quad \mathcal{D}_\alpha(\theta_1 \vee \theta_2)=\mathcal{D}_\alpha(\theta_1)\vee\mathcal{D}_\alpha(\theta_2) \quad \mathcal{D}_\alpha(\theta^{\star})=\mathcal{D}_\alpha(\theta)\cdot\theta^{\star}$$

$$\mathcal{D}_\alpha(ev)=\begin{cases} \epsilon & \mathit{if}\ \ \alpha\subseteq ev \\ \bot & \mathit{else} \end{cases} \quad \mathcal{D}_\alpha(Q)=\begin{cases} \epsilon & \mathit{if}\ \ \alpha=Q \\ \bot & \mathit{else} \end{cases} \quad \mathcal{D}_\alpha(\theta^{\infty})=\mathcal{D}_\alpha(\theta)\cdot\theta^{\infty}$$

$$\mathcal{D}_\alpha(\theta_1 \cdot \theta_2)=\begin{cases} (\mathcal{D}_\alpha(\theta_1)\cdot\theta_2)\vee\mathcal{D}_\alpha(\theta_2) & \mathit{if}\ \ \delta(\theta_1)=\mathit{true} \\ \mathcal{D}_\alpha(\theta_1)\cdot\theta_2 & \mathit{if}\ \ \delta(\theta_1)=\mathit{false} \end{cases} \quad \mathcal{D}_\alpha(\theta^{\omega})=\mathcal{D}_\alpha(\theta)\cdot\theta^{\omega}$$

5.1 Rewriting Rules

1. **Axioms.** Analogous to the standard propositional logic, $\bot$ (referring to *false*) entails any effects, while no *non-false* effects entails $\bot$.

$$\frac{\pi_1 \Rightarrow \pi_2}{\Omega \vdash (\pi_1, \bot) \sqsubseteq (\pi_2, \theta)}\ [\mathit{Bot\text{-}LHS}] \qquad \frac{\theta \neq \bot}{\Omega \vdash (\pi_1, \theta) \not\sqsubseteq (\pi_2, \bot)}\ [\mathit{Bot\text{-}RHS}]$$

2. **Disprove (Heuristic Refutation).** These rules are used to disprove the inclusions when the antecedent obviously contains more traces than the consequent. Here *nullable* and *infinitable* witness the empty trace and infinite traces respectively.

$$\frac{\delta(\theta_1)\wedge\neg\delta(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \not\sqsubseteq (\pi_2, \theta_2)}\ [\mathit{Dis\text{-}Nullable}] \qquad \frac{\varkappa(\theta_1)\wedge\neg\varkappa(\theta_2)}{\Omega \vdash (\pi_1, \theta_1) \not\sqsubseteq (\pi_2, \theta_2)}\ [\mathit{Dis\text{-}Infinitable}]$$

[6] *false* for unmentioned constructs.
[7] *false* for unmentioned constructs.
[8] $[\![\theta]\!]$ represents all the traces contained in θ.

3. **Prove.** We use the rule [*Reoccur*] to prove an inclusion when there exist inclusion hypotheses in the proof context Ω, which are able to soundly prove the current goal. One of the special cases of this rule is when the identical inclusion is shown in the proof context, we then prove it valid.

$$\frac{(\pi_1,\theta_1)\sqsubseteq(\pi_3,\theta_3) \in \Omega \quad (\pi_3,\theta_3)\sqsubseteq(\pi_4,\theta_4) \in \Omega \quad (\pi_4,\theta_4)\sqsubseteq(\pi_2,\theta_2) \in \Omega}{\Omega \vdash (\pi_1,\theta_1) \sqsubseteq (\pi_2,\theta_2)}\ [Reoccur]$$

4. **Unfolding (Induction).** This is the inductive step of unfolding the inclusions. Firstly, we make use of the auxiliary function *fst* to get a set of effects F, which are all the possible initial elements from the antecedent. Secondly, we obtain a new proof context Ω' by adding the current inclusion, as an inductive hypothesis, into the current proof context Ω. Thirdly, we iterate each element $\alpha \in F$, and compute the partial derivatives (*next-state* effects) of both the antecedent and consequent w.r.t α. The proof of the original inclusion succeeds if all the derivative inclusions succeed.

$$\frac{F = fst(\theta_1) \qquad \pi_1 \Rightarrow \pi_2 \qquad \forall\alpha \in F.\ (\theta_1\sqsubseteq\theta_2) :: \Omega \vdash D_\alpha(\theta_1) \sqsubseteq D_\alpha(\theta_2)}{\Omega \vdash (\pi_1,\theta_1) \sqsubseteq (\pi_2,\theta_2)}\ [Unfold]$$

Theorem 2 (TRS-Termination). *The rewriting system TRS is terminating.*

Proof. See Appendix C.

Theorem 3 (TRS-Soundness). *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns TRUE when proving $\Phi_1 \sqsubseteq \Phi_2$, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See Appendix D.

6 Implementation and Evaluation

To show the feasibility of our approach, we have prototyped our automated verification system using OCaml (See Zenodo [18]). The proof obligations generated by the verifier are discharged using Z3 [19]. We prove termination and soundness of the TRS. We validate the front-end forward verifier against the latest Multicore OCaml (4.12.0) implementation for conformance.

Table 1 presents the evaluation results of a microbenchmark, to demonstrate how verification scales with program size. We annotate 12 synthetic test programs with temporal specifications, half of which fail to verify. The experiments were done on a MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor. The table records: **No.**, the index of the program; **LOC**, lines of code; **Infer(ms)**, effects inference time; **#Prop(✓)**, number of valid properties; **Avg-Prove(ms)**, average proving time for the valid properties; **#Prop(✗)**, number of invalid properties; and **Avg-Dis(ms)**, average disproving time for the invalid properties.

Table 1. Experimental results.

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

Discussion: Generally, inference and proving time increase linearly with program length. Furthermore, we notice that disproving times for invalid properties are consistently lower than those for proved properties, regardless of program complexity. This finding echos the insights from prior TRS-based works [14,20–23], which suggest that TRS is a better average-case algorithm than those based on the comparison of automata.

A summary: A TRS is efficient because *it only constructs automata as far as it needs*, which makes it more efficient when disproving incorrect specifications, as we can disprove it earlier without constructing the whole automata. In other words, the more invalid inclusions are, the more efficient our solver is.

6.1 Case Studies

I. Encoding LTL. Classical LTL uses the temporal operators $\mathcal{G}$ ("globally") and $\mathcal{F}$ ("in the future"), which we also write $\Box$ and $\Diamond$, respectively; and introduced the concept of fairness, which places additional constraints on infinite paths. LTL was subsequently extended to include the $\mathcal{U}$ ("until") operator and the $\mathcal{X}$ ("next time") operator. As shown in Fig. 2, we encode these basic operators into our effects, making the specification more intuitive and readable, mainly when nested operators occur. Furthermore, by putting the effects in the precondition, our approach naturally subsumes *past-time LTL* along the way[9].

[9] Our implementation supports specifications written in LTL formulae, by providing a translator from LTL to *ContEffs*. The translation schema is taken from [17].

Table 2. Examples for converting LTL formulae into Effects. (l, j are labels.)

$\Box l \equiv l^\infty$	$\Diamond l \equiv \_^\star \cdot l$	$l \,\mathcal{U}\, j \equiv l^\star \cdot j$	$l \rightarrow \Diamond j \equiv \neg l \vee \_^\star \cdot j$
$\mathcal{X} l \equiv \_ \cdot l$	$\Box\Diamond l \equiv (\_^\star \cdot l)^\infty$	$\Diamond\Box l \equiv \_^\star \cdot l^\infty$	$\Diamond l \vee \Diamond j \equiv \_^\star \cdot l \vee \_^\star \cdot j$

II. Encoding Exceptions. Exceptions are a special case of algebraic effects which never resume, and Fig. 10 demonstrates how our framework soundly reasons about exceptions together with other kinds of effects. Here `raise()` performs `Exc` first, then does some other operations afterwards, represented by performing effect `Other`.

The handler on line 15 discharges `Exc` and returns, leaving the continuation `k` completely unused. Our fixpoint calculator computes the final trace of `excHandler` as simply `Exc`. We observe that the handler defined in the normal return (line 14) will be completely abandoned – because execution flow does not go back to `raise()` after handling `Exc`. The verified postcondition of `excHandler` matches how we would intuitively expect exceptions to work[10].

```
1  effect Exc: unit
2  effect Other: unit
3
4  let raise ()
5  (*@ req _^*  @*)
6  (*@ ens Exc!.Other! @*)
7  = perform Exc;
8    perform Other
9
10 let excHandler
11 (*@ req _^* @*)
12 (*@ ens Exc @*)
13 = match raise () with
14 | _ -> (* Abandoned *)
15 | effect Exc k -> ()
```

Fig. 10. Encoding Exceptions.

7 Related Work

Verification Framework: This work is a significant extension of [20,25], which deploys the verification framework, i.e., a forward verifier with a TRS. However, the goal of this paper is to reason about algebraic effects, which are octagonal and have different features from the sequential programs targeted in [20,25]. More specifically, our proposal handles coexistence of zero/one/multi-shot continuations; detects non-terminating behaviors; enforces static temporal properties of algebraic effects. None of these challenges has been tackled before.

Temporal Verification: One of the leading communities of temporal verification is automata-based model checking, mainly for finite-state systems. Various model checkers are based on some temporal logic specifications, such as LTL and CTL. Such tools extract the logic design from the program using modeling languages and verify specific assertions to guarantee various properties. Meanwhile,

[10] In general, each procedure has a set of circumstances for which it will terminate *normally*. An exception breaks the normal flow (these circumstances) of execution and executes a pre-registered exception handler instead [24].

to conduct temporal reasoning locally and for higher-order program, there is a sub-community whose aim is to support temporal specifications in the form of *effects* via the type-and-effect system. The inspiration from this approach is that it leads to a modular and compositional verification strategy, where temporal reasoning can be combined together to reason about the overall program [13,26,27]. However, the temporal effects in prior work tend to coarsely over-approximate the behaviors either via ω-regular expressions [26] or Büchi automata [27]. The conventional effects [13] have the form (Φ_u, Φ_v), which separates the finite and infinite effects. In this work, by integrating finite, infinite, and possibly both into a single disjunctive form, our effects eliminate the finiteness distinction, and enable an expressive modular temporal verification.

Type-and-effect Systems: Many languages with algebraic effects are equipped with type-and-effect systems – which enrich existing types with information about effects – to allow the effect-related behaviors of functions to be specified and checked. A common method of doing this is row-polymorphic effect types, used by languages such as Koka [6,28], Helium [29,30], Frank [31], and Links [32]. An effect *row* specifies a multiset of effects a function may perform, and is popular for its simplicity, expressiveness (naturally enabling *effect polymorphism*), and support for inference of principal effects [6]. There are numerous extensions to this model, including *presence types* attached to effect labels, allowing one to express the *absence* of an effect [32], existential and local effects for modularity [29], and linearity [33]. Other choices include sets of (instances of) effects [30], and structural subtyping constraints [34]. We consider finer-grained specifications of program behavior outside the realm of effect systems and discuss them separately.

Trace-based Effect Systems: Combining program events with a temporal program logic for asserting properties of event traces yields a powerful and general engine for enforcing program properties. Several works [35–37] have demonstrated that static approximations of program event traces can be generated by type and effect analyses [38,39], in a form amenable to existing model-checking techniques for verification. Trace-based analyses have been shown capable of statically enforcing flow-sensitive security properties such as safe locking behavior [40]; resource usage policies such as file usage protocols and memory management [37]; and enforcement of secure service composition [41].

More related to our work, prior research has been extending Hoare logic with event traces. Malecha et al. [42] focuses on finite traces (terminating) for web applications, leaving the divergent computation, which indicates *false*, verified for every specification. Nakata et al. [43] focuses on infinite traces (non-terminating) by providing coinductive trace definitions. Moreover, this paper draws similarities to *contextual effects* [44], which computes the effects that have already occurred as the prior effects. The effects of the computation yet to take place as the future effects. Besides, prior work [45] proposes an annotated type and effect system and infers behaviors from Concurrent ML [46] programs for channel-based communications, though it did not provide any inclusion solving process.

8 Conclusion

This work is mainly motivated by *how to modularly specify and verify programs in the presence of both user-defined primitive effects and effect handlers.*

To provide a practical proposal to verify such higher-order programs with crucial temporal constraints, we present a novel effects logic, *ContEffs*, to specify user-defined effects and effect handlers. This logic enjoys two key benefits that enable modular reasoning: the placeholder operator and the disjunction of finite and infinite traces. We demonstrate several small but non-trivial case studies to show *ContEffs*' feasibility. Our code and specification are particularly compact and generic; furthermore, as far as we know, this is the first temporal specification and proof of correctness of control inversion with the presence of algebraic effects.

Acknowledgement. We would like to thank the referees of APLAS 2022 for their helpful advice. This work is supported by grants NSOE-TSS2019-06 and MoE Tier-1 251RES1822.

References

1. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers. J. Log. Algebraic Methods Program. **84**(1), 108–123 (2015). https://doi.org/10.1016/j.jlamp.2014.02.001
2. Convent, L., Lindley, S., McBride, C., McLaughlin, C.: Doo bee doo bee doo. J. Funct. Program. **30**, e9 (2020). https://doi.org/10.1017/S0956796820000039
3. Hillerström, D., Lindley, S., Atkey, R.: Effect handlers via generalised continuations. J. Funct. Program. 30, e5 (2020). https://doi.org/10.1017/S0956796820000040
4. Sivaramakrishnan, K.C., Dolan, S., White, L., Kelly, T., Jaffer, S., Madhavapeddy, A.: 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, S. N. Freund and E. Yahav, Eds. ACM, pp. 206–221 (2021). https://doi.org/10.1145/3453483.3454039
5. Brachthäuser, J.I., Schuster, P., Ostermann, K.: Effekt: capability-passing style for type-and effect-safe, extensible effect handlers in scala. J. Funct. Programm. **30**, 1–46 (2020)
6. Leijen, D.: Koka: programming with row polymorphic effect types. In: Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014, ser. EPTCS, P. B. Levy and N. Krishnaswami, Eds., vol. 153, pp. 100–126 (2014). https://doi.org/10.4204/EPTCS.153.8
7. Landin, P.J.: A generalization of jumps and labels. Higher-Order Symbolic Comput. **11**(2), 125–143 (1998). https://doi.org/10.1023/A:1010068630801
8. De Vilhena, P.E., Pottier, F.: A separation logic for effect handlers. Proc. ACM Program. Lang. **5**, 1–28 (2021). https://doi.org/10.1145/3434314

9. Recursive cow. https://github.com/effect-handlers/effects-rosetta-stone/tree/master/examples/recursive-cow
10. Hillerström, D., Lindley, S.: Shallow effect handlers. In: Ryu, S. (ed.) APLAS 2018. LNCS, vol. 11275, pp. 415–435. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02768-1_22
11. Kammar, O., Lindley, S., Oury, N.: Handlers in action. ACM SIGPLAN Notices **48**(9), 145–158 (2013)
12. Antimirov, V.: Rewriting regular inequalities. In: Reichel, H. (ed.) FCT 1995. LNCS, vol. 965, pp. 116–125. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60249-6_44
13. Nanjo, Y., Unno, H., Koskinen, E., Terauchi, T.: A fixpoint logic and dependent effects for temporal property verification. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, ACM, pp. 759–768 (2018)
14. Antimirov, V.M., Mosses, P.D.: Rewriting extended regular expressions. Theor. Comput. Sci. **143**(1), 51–72 (1995). https://doi.org/10.1016/0304-3975(95)80024-4
15. Antimirov, V.: Partial derivatives of regular expressions and finite automata constructions. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 455–466. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-59042-0_96
16. Brotherston, J.: Cyclic proofs for first-order logic with inductive definitions. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 78–92. Springer, Heidelberg (2005). https://doi.org/10.1007/11554554_8
17. Leucker, M., Sánchez, C.: Regular linear temporal logic. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 291–305. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75292-9_20
18. Zenodo. https://zenodo.org/record/7009799#.Yw-yyuxBwRS
19. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
20. Song, Y., Chin, W.-N.: Automated temporal verification of integrated dependent effects. In: Lin, S.-W., Hou, Z., Mahony, B. (eds.) ICFEM 2020. LNCS, vol. 12531, pp. 73–90. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63406-3_5
21. Almeida, M., Moreira, N., Reis, R.: Antimirov and Mosses's rewrite system revisited. Int. J. Found. Comput. Sci. **20**(4), 669–684 (2009). https://doi.org/10.1142/S0129054109006802
22. Keil, M., Thiemann, P.: Symbolic solving of extended regular expression inequalities. In: 34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15–17, 2014, New Delhi, India, ser. LIPIcs, V. Raman and S. P. Suresh, Eds., vol. 29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 175–186 (2014). https://doi.org/10.4230/LIPIcs.FSTTCS.2014.175
23. Hovland, D.: The inclusion problem for regular expressions. J. Comput. Syst. Sci. **78**(6), 1795–1813 (2012). https://doi.org/10.1016/j.jcss.2011.12.003
24. Exception WiKi. https://en.wikipedia.org/wiki/Exception_handling
25. Song, Y., Chin, W.-N.: A synchronous effects logic for temporal verification of pure esterel. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 417–440. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_19

26. Hofmann, M., Chen, W.: Abstract interpretation from büchi automata. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), p. 51. ACM (2014)
27. Koskinen, E., Terauchi, T.: Local temporal reasoning. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), p. 59. ACM (2014)
28. Daan, L.: Type directed compilation of row-typed algebraic effects. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 486–499 (2017)
29. Biernacki, D., Piróg, M., Polesiuk, P., Sieczkowski, F.: Abstracting algebraic effects. In: Proceedings of the ACM on Programming Languages, vol. 3, no. POPL, pp. 1–28 (2019)
30. Dariusz, B., Maciej, P., Piotr, P., Filip, S.: Binders by day, labels by night: effect instances via lexically scoped handlers. In: Proceedings of the ACM on Programming Languages, vol. 4, no. POPL, pp. 1–29 (2019)
31. Lindley, S., McBride, C., McLaughlin, C.: Do be do be do. CoRR, vol. abs/1611.09259, (2016). http://arxiv.org/abs/1611.09259
32. Lindley, S., Cheney, J.: Row-based effect types for database integration. In: Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation, pp. 91–102 (2012)
33. Leijen, D.: Algebraic effect handlers with resources and deep finalization. Technical Report MSR-TR-2018-10. Tech. Rep, Microsoft Research (2018)
34. Pretnar, M.: Inferring algebraic effects. arXiv preprint arXiv:1312.2334 (2013)
35. Skalka, C., Smith, S., Van Horn, D.: Types and trace effects of higher order programs. J. Funct. Programm. **18**(2), 179–249 (2008)
36. Skalka, C., Smith, S.: History effects and verification. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 107–128. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30477-7_8
37. Marriott, K., Stuckey, P.J., Sulzmann, M.: Resource usage verification. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 212–229. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40018-9_15
38. Talpin, J.-P., Jouvelot, P.: The type and effect discipline. Inf. Comput. **111**(2), 245–296 (1994)
39. Amtoft, T., Nielson, H.R., Nielson, F.: Type and effect systems: behaviours for concurrency. World Sci. (1999)
40. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, pp. 1–12 (2002)
41. Bartoletti, M., Degano, P., Ferrari, G.L.: Enforcing secure service composition. In: 18th IEEE Computer Security Foundations Workshop (CSFW'05). IEEE, pp. 211–223 (2005)
42. Malecha, G., Morrisett, G., Wisnesky, R.: Trace-based verification of imperative programs with i/o. J. Symbolic Comput. **46**(2), 95–118 (2011)
43. Nakata, K., Uustalu, T.: A Hoare logic for the coinductive trace-based big-step semantics of while. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 488–506. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_26

44. Neamtiu, I., Hicks, M., Foster, J.S., Pratikakis, P.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 37–49 (2008)
45. Nielson, H.R., Amtoft, T., Nielson, F.: Behaviour analysis and safety conditions: a case study in CML. In: Astesiano, E. (ed.) FASE 1998. LNCS, vol. 1382, pp. 255–269. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053595
46. Reppy, J.H.: Concurrent ML: design, application and semantics. In: Lauer, P.E. (ed.) Functional Programming, Concurrency, Simulation and Automated Reasoning. LNCS, vol. 693, pp. 165–198. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56883-2_10