Abstract Interpretation of Temporal Safety Effects of Higher Order Programs

MIHAI NICOLA, Stevens Institute of Technology, USA
CHAITANYA AGARWAL, New York University, USA
ERIC KOSKINEN, Stevens Institute of Technology, USA
THOMAS WIES, New York University, USA

This paper describes a new *abstract interpretation*-based approach to verify temporal safety properties of recursive, higher-order programs. While prior works have provided theoretical impact and some automation, they have had limited scalability. We begin with a new automata-based "abstract effect domain" for summarizing context-sensitive dependent effects, capable of abstracting relations between the program environment and the automaton control state. Our analysis includes a new transformer for abstracting event prefixes to automatically computed context-sensitive effect summaries, and is instantiated in a type-and-effect system grounded in abstract interpretation. Since the analysis is parametric on the automaton, we next instantiate it to a broader class of history/register (or "accumulator") automata, beyond finite state automata to express some context-free properties, input-dependency, event summation, resource usage, cost, equal event magnitude, etc.

We implemented a prototype **ev**DRIFT that computes dependent effect summaries (and validates assertions) for OCaml-like recursive higher-order programs. As a basis of comparison, we describe reductions to assertion checking for higher-order but effect-free programs, and demonstrate that our approach outperforms prior tools DRIFT, RCAML/SPACER, MoCHI, and ReTHFL. Overall, across a set of 23 benchmarks, DRIFT verified 12 benchmarks, RCAML/SPACER verified 6, MoCHI verified 11, ReTHFL verified 18, and **ev**DRIFT verified 21; **ev**DRIFT also achieved a 6.3×, 5.3×, 16.8×, and 6.4× speedup over DRIFT, RCAML/SPACER, MoCHI, and ReTHFL, respectively, on those benchmarks that both tools could solve.

CCS Concepts: • **Theory of computation → Automated reasoning**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: temporal verification, type-and-effect systems, dependent temporal effects

ACM Reference Format:
Mihai Nicola, Chaitanya Agarwal, Eric Koskinen, and Thomas Wies. 2025. Abstract Interpretation of Temporal Safety Effects of Higher Order Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 362 (October 2025), 29 pages. https://doi.org/10.1145/3763140

1 Introduction

The long tradition of temporal property verification has, in recent years, been also directed at programs written in languages with recursion and higher-order features. In this direction, a first step was to go beyond simple types to dependent and/or refinement type systems [13, 49, 52, 62, 66], capable of validating merely (non-temporal) safety assertions. Subsequently, works focused on verifying termination of higher-order programs, e.g., [38].

As a next step, researchers focused on *temporal* properties of higher-order programs. In this setting, programs have a notion of observable *events* or *effects*, typically emitted as a side effect of a

Authors' Contact Information: Mihai Nicola, Stevens Institute of Technology, Hoboken, USA, lnicola@stevens.edu; Chaitanya Agarwal, New York University, New York, NY, USA, ca2719@nyu.edu; Eric Koskinen, Stevens Institute of Technology, Hoboken, USA, eric.koskinen@stevens.edu; Thomas Wies, New York University, New York, NY, USA, wies@cs.nyu.edu.

ACM 2475-1421/2025/10-ART362
https://doi.org/10.1145/3763140

program expression such as "ev e," where e is first reduced to a program value and then emitted. The semantics of the program is correspondingly augmented to reduce to a pair (v, π), where v is the value and π is a sequence of events or an "event trace." For such programs a natural question is whether the set of all event traces is included within a given temporal property expressed in Linear Temporal Logic [50], or as an automaton. Liveness properties apply to programs that may diverge, inducing infinite event traces. A first approach at automated temporal verification was through the celebrated reduction to fair termination [64]. Murase et al. [43] introduced a reduction from higher-order programs and LTL properties to termination of a calling relation.

In a parallel research trend, others have been exploring compositional type-and-effect theories for temporal verification. Skalka and Smith [56] and Skalka et al. [57] described a type-and-effect system to extract a finite abstraction of a program and then perform model-checking on that abstraction. Later, Koskinen and Terauchi [36] and Hofmann and Chen [21] showed that the effects component in a type-and-effect system $\Gamma \vdash e : \tau \& \varphi$ could consist of a temporal property φ, where φ holds of the events generated by the reduction of expression e. This was combined with a dependent refinement system by Koskinen and Terauchi [36] and used with an abstraction of Büchi automata by Hofmann and Chen [21]. Nanjo et al. [44] then later gave a deductive proof system for verifying such temporal effects, even permitting the temporal effect expressions to depend on program inputs. In a more distantly related line of research, others consider languages with programmer-provided "algebraic effects" and their handlers [39, 54] (see Sec. 9).

1.1 Better Automation Through Abstract Interpretation

The first step of this paper is a new route to automate temporal effect inference and verification of recursive higher-order programs through abstract interpretation.

As a preliminary step, we describe a direct approach that reduces verification of such effect-full programs to verifying assertions of effect-less higher-order programs. We later experimentally show that, although this theoretically enables higher-order safety verifiers (e.g. Drift, RCaml/Spacer, MoCHi, and ReTHFL) to be applied to the effect setting, those tools do not exploit much of the property structure and ultimately struggle on the inherent overhead that comes from these transformations.

To achieve a more scalable solution, our core contribution is a novel *effect abstract domain*. In the concrete semantics, an execution is simply the program execution environment paired with the event trace prefix that was thus far generated, i.e., an element of $(\mathcal{V}^* \times Env)$ where $\mathcal{V}$ is the domain of program values and Env the domain of value environments. We first observe that both the environment and the possible trace prefix, somewhat counterintuitively, can be organized around the automaton control state. That is, an abstraction like $Q \to \wp(Env)$ captures the possible execution environments that could be reachable at a control state $q \in Q$ of the automaton. This control state-centric summary of environments enables the abstract domain to capture disjunctive invariants, guided by the target property of the verification. This abstraction often avoids the need for switching to a more expensive abstract domain that is closed under precise joins. Having organized around control state, the final abstraction step is to associate with each q a summary of the program environment, e.g. constraints like `x > y`. The abstract domain for summarizing program environments can naturally be instantiated using any of a variety of standard numerical domains such as polyhedra [2, 7, 55], octagons [42], etc.

1.2 Better Expressiveness Through Accumulator Automata

The effect abstract domain above turns out to be somewhat parametric over the kind of automaton, opening up another opportunity.

Specifically, there are many temporal safety properties that go beyond basic event sequencing properties especially, for example, if each event emits an integer. Examples include a property that the sum of the emitted integers is below some bound (e.g. resource analysis), or that the last emitted integer is the largest one. Properties could depend on inputs (see, e.g. Nanjo et al. [44]), or involve context-free-like properties such as protocols of stateful APIs [11] or the sum of production being equal to the sum of consumption.

We support this wider class of temporal safety properties by augmenting our effect abstract domain automata to symbolic accumulator automata (SAA). Our automaton model is inspired by the various notions of (symbolic) register or memory automata considered [3, 9, 25] and consists of a register "accumulator" (e.g., an integer or tuple of integers) that can remember earlier events, calculate summaries, etc. SAA is expressive enough to capture the example properties above.

To instantiate SAA in our framework, we refine the effect abstraction to $Q \rightarrow \wp(\mathcal{V} \times Env)$, now capturing the possible *pairs of* accumulator value and execution environment that could be reachable at control state $q \in Q$. Our abstraction thus associates with each q: (i) a summary of the program environment, e.g. constraints like $\mathsf{x} > \mathsf{y}$, (ii) a summary of the automaton accumulator, e.g. constraints like $\mathsf{acc} > 0$, and even (iii) relations between the two, e.g. $\mathsf{acc} > \mathsf{x} - \mathsf{y}$. Thus, in this example, we capture at location ℓ in the program, that control state q is reachable but only in a configuration where the accumulator is positive, the program variable x is greater than y and the accumulator bounds the difference between x and y.

1.3 Challenges & Contributions

To pursue the effect abstract domain, we address the following challenges in this paper:

Accumulative type and effect system (Sec. 4). Our effect abstract domain, expressing properties of program expressions, is associated with the program through a type-and-effect system, with judgments of the form $\Gamma\ ;\ \phi \vdash e : \tau \& \phi'$, where ϕ summarizes the prefix up to the evaluation of e, ϕ' summarizes the *extended* prefix with the evaluation of e, and term-specific premises dictate how extensions are formed. The system is parametric in the abstract domains used to express dependent effects and dependent type refinements. Our system resembles existing systems for sequential effects such as [15, 16] but is grounded in abstract interpretation to facilitate automated inference of types and effects.

Effect abstract domain (Sec. 5). We formalize the abstract domain discussed above as an instantiation of our effect system. A key ingredient is the *effect extension operator* $\odot$ that takes an abstraction of a reachable automaton configuration ϕ, a type of a new event β (we use refinement types for β to capture precise information about the possible values of the event to extend a trace prefix), and produces an abstraction of the automaton configurations reachable by the extended trace. The user-provided automata include symbolic error state conditions and so if the effect computed by the analysis associates error states with bottom, then the property encoded by the automaton holds of the program. Finally, we have proved the soundness of the effect abstract domain.

Automated inference of effects (Sec. 6). We next address the question of automation. Recent work showed that, for programs *without effects*, that abstract interpretation can be used to compute refinement types through a higher-order dataflow analysis [49]. We present an extension to *effectful* programs through a translation-oriented embedding of programs with effects to effect-free programs and a specialized abstract transformer that exploits the structure of the translated programs and effect abstract domain. The resulting abstract interpretation propagates effects in addition to values through the program. To obtain the overall soundness of the inference algorithm, we show that the types inferred for the translated programs can be used to reconstruct a derivation in our type and effects system.

Verification, Implementation & Benchmarks (Sec. 7). We implement the type system, effect abstract domain and abstract interpretation in a new tool **ev**Drift for OCaml-like recursive higher-order programs. Our implementation is an extension of the Drift tool, which provides assertion checking of effect-free programs. There are no existing tools that can verify SAA properties of higher-order event-generating programs. Thus, in an effort to find the closest basis for comparison, we also implemented a translation that reduce SAA verification of effect programs to assertion checking of effect-free programs (to which Drift, RCaml/Spacer, MoCHi, ReTHFL, etc. can be applied). To improve the precision of our abstract interpretation, we further adapted the classical notion of *trace partitioning* [41] to this higher-order effect setting.

To date there are limited higher-order benchmark programs with properties that require an automaton with a register to express. We thus built the first suite of such benchmarks by creating 23 new examples and adapting examples from the literature including summation/max-min examples [3, 9, 25], monotonicity examples, programs with temporal event sequences [36, 43, 44], resource analysis [19, 20, 22], and an auction smart contract [59].

Evaluation (Sec. 8). We evaluated (i) the effectiveness of **ev**Drift at directly verifying SAA-expressible temporal safety properties over the use of Drift, RCaml/Spacer, MoCHi, and ReTHFL when applied via the translation/reduction to assertion checking, and (ii) the degree to which trace partitioning improves precision for **ev**Drift. Overall, our approach is able to verify 21 out of the 23 benchmarks, which is 9, 15, 10, and 3 more than Drift, RCaml/Spacer, MoCHi, and ReTHFL, respectively, (with our tuple translation) could verify. Furthermore, **ev**Drift achieved a speedup of 6.3×, 5.3×, 16.8×, and 6.4× over Drift, RCaml/Spacer, MoCHi, and ReTHFL, respectively, on those benchmarks that both tools could solve. *The supplement to this paper includes the* **ev**Drift *source, all benchmark sources, and the Appendix.*

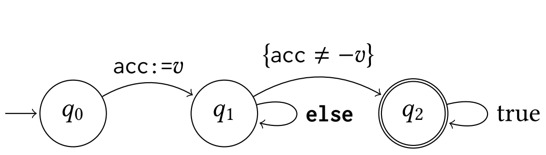
2 Overview

This paper introduces a method for verifying properties of dependent effects of higher-order programs, through an abstraction that can express relationships between the (symbolic) next step of an automaton and the dependent typing context of the program at the location where a next event is emitted. We show that, when combining our approach with data-flow abstract interpretation [49], and an abstract domain of symbolic accumulator automata, we can verify a variety of memory-based, dependent temporal safety properties of higher-order programs.

2.1 Motivating Examples

Example 2.1. Consider the following example:

```
let rec busy n t =
  if (n <= 0) then ev (-t)
  else busy (n - 1) t
let main (x:int) (n:int) =
  ev x; busy n x
```



Above in main, an integer event x is emitted, and then a recursive function busy repeatedly iterates until n is below 0, at which point the event -t (which is equal to -x) is emitted. For this program, the possible event traces are simply $\{x; -x \mid x \in \mathbb{Z}\}$, i.e., any two-element sequence of an integer and its negation. This property can be expressed by a *symbolic accumulator automaton* (a cousin to symbolic automata and to memory automata, as discussed in Sec. 5), as shown above. The automaton is provided by the user along with the program. It has an initial control state q_0, from which point, whenever an event $\mathsf{ev}(v)$ is observed for any integer v, the automaton's internal register acc is updated to store value v and a transition is taken to q_1. From q_1, observing another event whose value is not the negation of the saved acc will cause a transition to the final accepting state q_2 or otherwise loop at q_1. The language of the automaton consists of traces that violate the property of interest. That is, the property expressed by the automaton is the complement of the automaton's language. It consists of the traces: $\{x(-x)^* \mid x \in \mathbb{Z}\}$, which permits none or arbitrarily many $-x$ events after x. We note that a stronger specification that exactly characterizes the set $\{x; -x \mid x \in \mathbb{Z}\}$ can also be expressed (and verified with our approach). We use the weaker specification here to highlight that, in general, the automaton approximates the program's traces.

Direct approach: reduction to assertion checking. At least in theory, this program/property can be verified using existing tools through a cross-product transformation between the program and property that reduces the problem to an assertion-checking safety problem. As is common, the automaton can be encoded in the programming language (or the program can be converted to an automaton [18]) with integer variables q and acc for the automaton's control state and accumulator, respectively. The automaton's transition function is also encoded in the language through simple if-then-else expressions. This is shown in the function ev_step, which consumes the current automaton configuration, and a next event value v and returns the next configuration:

```
let ev_step q acc v : (Q * int) =
  (* take one automaton step *)
  if      (q==0) then (1, v)
  else if (q==1 && v==-acc) then (2,acc)
  else if (q==1) then (1,acc)
  else (q,acc)
```

A product can then be formed, for example, by passing and returning the (q,acc) configuration into and out of every expression, and replacing **ev** expressions (which are not meaningful to typical safety verifiers) with a call to ev_step. For Ex. 2.1, this yields the following product program:

```
let rec busy_prod q acc n t =
  if (n <= 0) then ev_step q acc (-t)
  else busy_prod q acc (n - 1) t
```

```
let main_prod (x:int) (n:int) =
  let (q,acc) = (0,0) in
  let (q',acc') = ev_step q acc x in
  let (q'',acc'') = busy_prod q acc n x
  in assert(q''==2)
```

In main_prod above, the initial configuration is provided for the automaton, then the first event expression is replaced by a call to ev_step, then the resulting next configuration is passed to busy_prod and the returned final configuration is input to an **assert**. busy_prod is similar.

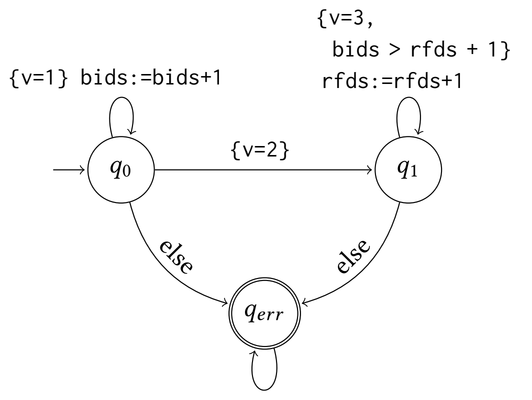
We implemented the above translation (details in the extended version [46]) and used it in combination with a variety of existing verification tools for event-less higher-order programs: (1) the Drift tool which uses a dependent type system and abstract interpretation to verify safety properties of higher-order recursive programs [49], (2) RCaml/Spacer (part of CoAR[63]), another fairly mature tool that can also verify assertions of higher order programs [30, 37, 54], (3) MoCHi [53], another software model checker based on higher-order recusion schemes [32, 33], and (4) ReTHFL, a type-based validity checker for a fragment of a higher-order fixed-point logic, that leverages CHC solvers to infer predicates within a refinement type system.

The problem. Although this example tuple product reduction can be verified by these existing tools, unsurprisingly, the approach does not scale well with any of the considered tools. Let us examine another example called auction, shown in the top left of Fig. 1, that is only slightly more involved yet already demonstrates the problem for existing tools when the tuple product reduction is used: Drift reports a potential assertion violation after 55.1 s, RCaml/Spacer times out after 900 s, and MoCHi reports a potential assertion violation after 91 s. Only ReTHFL can verify the

Input Program:

```
1 let refund k kamt h _ =
2   if k <= 1 then ()
3   else ((ev 3)ⓡ; h ())
4 let close j g =
5   if j = 1 then ()
6   else ((ev 2)ⓒ; g ())
7 let rec bid i iamt f =
8   let nmax = iamt + 1 in
9   if * then
10    ((ev 1)ⓑ;
11    bid (i+1) nmax (refund i iamt f))
12  else close i f
13 let main () = (bid 1 1 (λ _.()))ⓕ
```

Input Property: Initially, `bids`= 0 and `rfds`= 0.



Computed Effect Abstractions :

Location ⓑ :
$q_0^{\textcircled{b}} \mapsto (\texttt{bids} = \texttt{i}) \land (\texttt{i} >= 1)$
$q_1^{\textcircled{b}} \mapsto \bot$
$q_{err}^{I\textcircled{b}} \mapsto \bot$

Location ⓒ :
$q_0^{\textcircled{c}} \mapsto \bot$
$q_1^{\textcircled{c}} \mapsto \texttt{bids} = (\texttt{j} - 1) \land (\texttt{j} >= 2) \land (\texttt{rfds} = 0)$
$q_{err}^{I\textcircled{c}} \mapsto \bot$

Location ⓡ :
$q_0^{\textcircled{r}} \mapsto \bot$
$q_1^{\textcircled{r}} \mapsto \texttt{bids} = (\texttt{rfds} + \texttt{k} - 1) \land (\texttt{k} >= 2)$
$q_{err}^{I\textcircled{r}} \mapsto \bot$

Location ⓕ :
$q_0^{\textcircled{f}} \mapsto (\texttt{bids} = 0) \land (\texttt{rfds} = 0)$
$q_1^{\textcircled{f}} \mapsto \texttt{bids} = \texttt{rfds} + 1$
$q_{err}^{I\textcircled{f}} \mapsto \bot$

For every other location ⓘ : $q_{err}^{\textcircled{i}} \mapsto \bot$

Fig. 1. Top left shows the **auction** example. Top right illustrates the SSA property. At the bottom is the computed effects inferred by our tool.

example, but still needs 18.6 s. We will describe a technique and tool that can instead verify this example in only 2.7 s. In fact, as we will see in our evaluation (§8), for several more elaborate benchmarks like those inspired by amortized complexity analysis, this techniques is the only one for which verification succeeds.

The `auction` example in Fig. 1 involves a first stage in the `bid` function in which some nondeterministic number of bidders place increasing bids. Each bid event is represented as an **ev** `1` event (Ln 10). Then, a `close` event **ev** `2` occurs (Ln 6), after which point, the `k-1` losers are refunded as a `refund` event **ev** `3` (Ln 3). This recursive program is also **higher-order**: `bid` constructs a function `(refund i iamt f)` that tracks the amount `iamt` to be refunded to bidder `i` that was overtaken by the new bid, and `f` is a similar function that tracks all previous refunds. When the bidding closes, the last constructed refund function is called to apply all refunds.

The event traces of the program are: $\{(1^n; 2; 3^{n-1}) \mid n \in \mathbb{N}\}$, i.e. any sequence of some n number of "1"-events, followed by a "2"-event, followed by $n-1$ occurrences of "3"-events. A simple temporal safety property, expressed as an automaton, that ensures the correct *order* of events could involve three states: an initial state q_0 that loops at bid "1"-events, a transition under close "2"-events to an accepting state q_1, self-loop to q_1 under refund "3"-events, and otherwise transitions to error state q_{err}. These states and transitions are depicted in the top right of Fig. 1. With an accumulator automaton, this property can be improved to more accurately capture the valid trace histories by counting the bids: we use a tuple accumulator (`bids,rfds`) that has a counter for the number of `bids` and a counter for the number of refunds `rfds`. The self-loop at q_0 increments `bids`, and the self-loop at q_1 ensures that more refunds have not been given than bids, and increments `rfds`.

The struggle. A translation-based reduction to existing safety verification tools for higher-order programs does not fare well and the reason is twofold. First, there is a blowup in the size of the analyzed program due to the translation, which causes a significant increase in analysis time. In addition, tools like Drift use abstract domains that are not closed under arbitrary disjunctions. A translation of the automaton's state space and transition relation into the program will cause loss of precision due to computation of imprecise joins at data-flow join points. This will cause the analysis to infer an effect abstraction that is too imprecise for verifying the desired property.

2.2 Effect Abstract Domain

The key idea of this paper is to exploit the structure of the automaton to better capture disjunctive reasoning in the abstract domain. Roughly speaking, the abstract domain will associate each *concrete* automaton control state q, with *abstractions* of (i) the event sequences that could lead to q and (ii) the possible program environment at q. This abstraction is expressed as a relation between the accumulator value and the program environment. We will now describe this abstraction and see the resulting computed abstraction depicted in the bottom of Fig. 1.

We obtain this abstraction in three main steps, provided a given input symbolic accumulator automaton $A = (Q, \mathcal{V}, \delta, \mathsf{acc}, \ldots)$ with the alphabet being some set of values $\mathcal{V}$ (in this section let $\mathcal{V} = \mathbb{Z}$) and transitions updating the control state and accumulator. We now discuss these steps.

Concrete semantics. To begin, the concrete semantics of the program is simply pairs of event traces $\mathbb{Z}^*$ with program environments, i.e., $\wp(\mathbb{Z}^* \times Env)$. Transitions in the concrete semantics naturally update the environment in accordance with the reduction rules, and the event sequence is only updated when an expression $\textbf{ev}\ v$ is reduced: $\wp(\mathbb{Z}^* \times Env) \xrightarrow{\textbf{ev}\ v} \wp(\mathbb{Z}^* \times Env)$. For the above example, a concrete sequence of states and transitions could be the following:

$$
\begin{aligned}
&(\epsilon, [main, \text{(empty env)}]) \rightsquigarrow (\epsilon, [bid, \mathtt{i}:1, \mathtt{iamt}:1, \mathtt{f}:(\lambda\ \_\ldots)])\\
&\xrightarrow{\textbf{ev}\ 1} (\{1\}, [bid, \mathtt{i}:1, \mathtt{iamt}:1, \mathtt{f}:(\lambda\ \_\ldots)]) \overset{(\textbf{ev}\ 1)^{41}}{\rightsquigarrow} (\{1^{42}\}, [close, \mathtt{j}:43, \mathtt{g}:(\lambda\ \_.(\lambda\ \_\ldots))])\\
&\xrightarrow{\textbf{ev}\ 2} (\{1^{42}, 2\}, [refund, \mathtt{k}:42, \mathtt{kamt}:42, \mathtt{h}:(\lambda\ \_.(\lambda\ \_\ldots))]\\
&\xrightarrow{\textbf{ev}\ 3} (\{1^{42}, 2, 3\}, [refund, \mathtt{k}:42, \mathtt{kamt}:42, \mathtt{h}:(\lambda\ \_.(\lambda\ \_\ldots))])\\
&\overset{(\textbf{ev}\ 3)^{40}}{\rightsquigarrow} (\{1^{42}, 2, 3^{41}\}, [refund, \mathtt{k}:2, \mathtt{kamt}:2, \mathtt{h}:(\lambda\ \_\ldots))])
\end{aligned}
$$

(Technically a transition takes the powerset of possible sequence/environment pairs to another powerset; here we show only one sequence for simplicity.) Above the first component is an event sequence, starting with the empty sequence ϵ and, for this nondeterministic behavior, the trace will accumulate the event sequence $1^{42}; 2; 3^{41}$.

Intermediate abstraction via concrete automaton control states. With integer variables and integer effect sequences, it is clear that abstraction is needed to represent the possible event sequences of a program even as simple as this running example. In this example, there are infinitely many sequences of the form $1^k; 2; 3^{k-1}$. The first key idea we explore in this paper is to organize the abstraction around the automaton and, crucially, *keep the automaton control state concrete* while abstracting everything else: the environment, the possible event sequence prefixes, and the value of the automaton's accumulator. The benefit is that this will lead to a somewhat disjunctive abstract effect domain, where event trace prefixes can be categorized according to the control state (and accumulator values and program environments) that those prefixes reach. To this end, the first layer of abstraction uses the automaton control states Q (rather than merely event sequences), and associates each automaton control state with the possible set of pairs of accumulator value $\mathbb{Z}$ and program environment that reach that state along some event sequence: $Q \mapsto \wp(\mathbb{Z} \times Env)$. At this layer, transitions from an expression **ev** v are captured through the automaton's transition function $\delta(v)$, which leads to a (possibly) new automaton state and updates the accumulator value: $Q \mapsto \wp(\mathbb{Z} \times Env) \xrightarrow{\delta(v)} Q \mapsto \wp(\mathbb{Z} \times Env)$. For the `auction` example, when an execution iterates `bid` 42 times, there is an event trace prefix 1^{42}, then the following lists some of the effects at body of `bid` per each q:

$$q_0 \mapsto \{(\{1\}, (\texttt{i}:1, \texttt{iamt}:1, \texttt{f}:\_)), (\{1, 1\}, (\texttt{i}:2, \texttt{iamt}:2, \texttt{f}:\_)), \ldots\}, \quad q_1 \mapsto \emptyset, \quad q_{err} \mapsto \emptyset.$$

Above q_1 is not reachable yet because at the point when the program reaches location ⓒ, at least one close ("2") event must have been emitted. Similarly q_{err} is not yet reachable. q_0 is, however, reachable with event sequences of the form 1^k and in the corresponding environment `i` will be equal to k.

Abstract relations with the accumulator. Thus far we associate event sequence and environment pairs per control state, but there are still infinite sets of pairs. We thus next abstract *relations* between the accumulator values at location q and the environments, employing a parametric abstract domain of base refinement types. That is, the type system provides abstractions of program values, which we can then also relate to abstractions of the accumulator. We will discuss the formal details of this abstraction in Sec. 5 but illustrate the abstraction in the bottom of Fig. 1. For every location ⓘ and automaton state q_j, we compute a summary of the possible trace prefixes and corresponding abstraction of the program variables, accumulator, and relations between them. In this example, at the **ev** `1` location denoted ⓑ, our summary for $q_0^{ⓑ}$ reflects that the number of bid (1) events in the prefix counted by accumulator `bid` is equal to the environment variable `i`, and that `i` is positive. No other automaton states are reachable. Meanwhile, at the **ev** `3` location denoted ⓡ, our summary for $q_1^{ⓡ}$ reflects that the number of refund (3) events seen in the prefix so far is `k` − 1 away from the number of bid (1) events, and that `k` ≥ 2. The automaton specifies if ever this is violated it will transition to q_{err}. The program is safe because at every location ⓘ, we compute $q_{err}^{ⓘ} \mapsto \bot$. Our accumulator automata can also include assertions that can be applied at the end of a trace. In this example, we would like to prove that the number of refunds was one less than the number of bids. We also compute abstractions at the final program location denoted ⓕ, including the fact that `bids=rfds+1` (or `bids=rfds=0`), which validates the end-of-program assertion.

2.3 Type System, Inference, Evaluation

Our approach to verifying effects is fully automated. Toward achieving this, the rest of this paper addresses the challenges identified in Sec. 1, but here with more detail in the context of this example:

Accumulative type and effect system (Sec. 4). In order to form relations between reachable automaton configurations' accumulator and program variables, we present a novel dependent type and effect system that is *accumulative* in nature. The type system allows us to, for example, express judgments on the (**ev** `3`)$^{ⓡ}$ expression to ensure that the count of bid (2) events is at least one more than the count of refund (3) events.

First, let ϕ_{acc}^{auct}(`k`) be shorthand for `bids = rfds + k - 1` ∧ `k >= 2` , i.e., that the accumulator count of bids is equal to the accumulator count of refunds plus program variable `k` minus one, and that the value of `k` is at least 2. Further, due to the nested construction of delayed refund calls with decreasing arguments, when we reach (**ev** `3`), we have that $k \rightsquigarrow (k-1)$. We thus obtain the judgment below. We focus on the boxed area, in which we compute the abstract effect concatenation operation denoted $\odot$. This concatenation is between q_1's existing effect in the context along with path condition k>1, and the $g$uard/$u$pdate for a next single refund event (3).

$$\begin{array}{l}\Gamma; [q_0 \mapsto \bot, q_1 \mapsto \phi^{auct}_{acc}(\texttt{k+1}), q_{err} \mapsto \bot, \ldots] \\ \vdash \textbf{ev}\ 3 : ()\& \left[\begin{array}{l} q_0 \mapsto \bot, \\ q_1 \mapsto \boxed{[\phi^{auct}_{acc}(\texttt{k+1}) \wedge (\texttt{k > 1})] \odot [g : \texttt{bids > rfds + 1}]; [u : \texttt{rfds := rfds + 1}]}, \\ q_{err} \mapsto \bot \end{array}\right]\end{array}$$

The context information $\phi^{auct}_{acc}(\texttt{k+1})$ is strengthened by the constraints on the program variable k imposed by the branching condition and this is sufficient to ensure the validity of the transition guard bids > rfds + 1. The update of the accumulator rfds to rfds+1 reestablishes $\phi^{auct}_{acc}(\texttt{k})$ at q_1. Moreover, the result of the concatenation guarantees that q_{err} remains unreachable.

Effect abstract domain (Sec. 5). We formalize the effect abstract domain discussed above.

Automated inference of effects (Sec. 6). We introduce a dataflow abstract interpretation inference of types that calculates summaries of effects, organized around concrete automaton control states, as seen in the example in Fig. 1. To achieve this, we exploit the parametricity of type systems (like [49]) over the kinds of constructs in the language, introducing *sequences* as a new base type. We then embed sequences into the q-indexed effect components.

Verification, Implementation & Benchmarks (Sec. 7). To verify examples like auction (and others among the 23 benchmarks), we have implemented our (i) abstract effect domain, (ii) accumulative type and effect system and (iii) automated inference in a new tool called **ev**Drift. **ev**Drift takes, as input, the program in an OCaml-like language (Fig. 1) as well as a symbolic accumulation automaton, written in a simple specification language (control states and the accumulator are integers and the automaton transition function is given by **ev**Drift expressions). In Sec. 7 we discuss how our inference is used for verification, and implement the product reductions to compare against tools for effect-free programs.

Evaluation (Sec. 8). **ev**Drift verifies auction in 2.7s, whereas previous assertion-verifiers (combined with our translations) either took significantly longer to verify it (18.6s for ReTHFL), timeout (RCaml/Spacer) or fail to verify (Drift and MoCHi). More generally, **ev**Drift verifies more examples and otherwise outperforms Drift, RCaml/Spacer, MoCHi, and ReTHFL by 6.3×, 5.3×, 16.8×, 6.4× resp. on benchmarks that each solve.

3 Preliminaries

We briefly summarize background definitions and notation. The formal development of our approach uses an idealized language based on a lambda calculus with terms $e \in \mathcal{E} ::= c \mid x \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \lambda x.\ e \mid (e\ e) \mid \textbf{ev}\ \ e$ and values $v \in \mathcal{V} ::= c \mid \lambda x.\ e$. Expressions e in the language consist of constant values $c \in \mathit{Cons}$ (e.g. integers and Booleans), variables $x \in \mathit{Var}$, function applications, lambda abstractions, conditionals, and event expressions $\textbf{ev}\ \ e_1$. We assume the existence of a dedicated unit value $\bullet \in \mathit{Cons}$ and the Boolean constants $\texttt{true}, \texttt{false} \in \mathit{Cons}$. Values $v \in \mathcal{V}$ consist of constants and lambda abstractions. We will often treat expressions as equal modulo alpha-renaming and write $e[e'/x]$ for the term obtained by substituting all free occurrences of x in e with term e' while avoiding variable capturing. We further write $\mathrm{fv}(e)$ for the set of free variables occurring in e.

A *value environment* ρ is a total map from variables to values: $\rho \in \mathit{Env} \overset{\text{def}}{=} \mathit{Var} \to \mathcal{V}$.

The operational semantics of the language is defined with respect to a transition relation over configurations $\langle e, \pi \rangle \in \mathcal{E} \times \mathcal{V}^*$ where e is a closed expression representing the continuation and π is a sequence of values representing the events that have been emitted so far. All configurations are considered initial and configurations $\langle v, \pi \rangle$ are terminal. To simplify the reduction rules, we use evaluation contexts E that specify evaluation order: $E ::= [] \mid E\ e \mid v\ E \mid \texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \textbf{ev}\ E$.

$$\textsc{e-app}\quad \langle(\lambda x.e)\ v, \pi\rangle \longrightarrow \langle e[v/x], \pi\rangle \qquad \textsc{e-ev}\quad \langle \mathbf{ev}\ v, \pi\rangle \longrightarrow \langle \bullet, \pi \cdot v\rangle \qquad \textsc{e-context}\ \frac{\langle e, \pi\rangle \longrightarrow \langle e', \pi'\rangle}{\langle E[e], \pi\rangle \longrightarrow \langle E[e'], \pi'\rangle}$$

$$\textsc{e-ite-true}\quad \langle \mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2, \pi\rangle \longrightarrow \langle e_1, \pi\rangle \qquad \textsc{e-ite-false}\quad \langle \mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2, \pi\rangle \longrightarrow \langle e_2, \pi\rangle$$

Fig. 2. Reduction rules of operational semantics.

The transition relation $\langle e, \pi\rangle \longrightarrow \langle e', \pi'\rangle$ is then defined in Fig. 2. In particular, the rule E-EV captures the semantics of event expressions: the evaluation of **ev** v returns the unit value and its effect is to append the value v to the event sequence π. We write $\langle e, \pi\rangle \rightsquigarrow \langle e', \pi'\rangle$ to mean that $\langle e, \pi\rangle \longrightarrow^* \langle e', \pi'\rangle$ and there exists no $\langle e'', \pi''\rangle$ such that $\langle e', \pi'\rangle \longrightarrow \langle e'', \pi''\rangle$.

(Non-accumulative) type and effect systems. Conventional type and effect systems [40] typically take the form $\Gamma \vdash e : \tau \& \phi$ and capture the local effects that occur during the evaluation of expression e to value v. Such systems have also been extended to the setting of higher-order programs [44, 56, 57]. While these systems are generally suitable to deductive reasoning, the judgements assume no information describing the program's behavior up to the evaluation of the respective expression. They thus fail to provide contextual reasoning for effects and so they suffer from a lack of precision and increase the difficulty of automation.

4 Accumulative Type and Effect System

In this section, we present an abstract formalization of our dependent type and effect system for checking accumulative effect safety properties. The notion is parameterized by the notion of basic refinement types, which abstract sets of constant values, and the notion of dependent effects, which abstract sets of event sequences. Both abstractions take into account the environmental dependencies of values and events according to the context where they occur in the program. To facilitate the static inference of dependent types and effects, we formalize these parameters in terms of abstract domains in the style of abstract interpretation.

Base refinement types. We assume a lattice of base refinement types $\langle \mathcal{B}, \sqsubseteq^b, \bot^b, \top^b, \sqcup^b, \sqcap^b\rangle$. Intuitively, a basic refinement type $\beta \in \mathcal{B}$ represents a set of pairs $\langle c, \rho\rangle$ where $c \in \mathit{Cons}$ and $\rho \in \mathit{Env}$ is a value environment capturing c's environmental dependencies. To formalize this intuition, we assume a *concretization function* $\gamma^b \in \mathcal{B} \to \wp(\mathcal{V} \times \mathit{Env})$. We require that γ^b is monotone and top-strict (i.e., $\gamma^b(\top^b) = \mathcal{V} \times \mathit{Env}$). We assume the existence of a basic refinement type *bool* such that $\gamma^b(\mathit{bool}) = \{\mathbf{true}, \mathbf{false}\} \times \mathit{Env}$.

We let $\mathrm{dom}(\beta)$ denote the set of variables $x \in \mathit{Var}$ that are constrained by β. Formally:

$$\mathrm{dom}(\beta) = \{\, x \in \mathit{Var} \mid \exists v, \rho, \rho'. \langle v, \rho\rangle \in \gamma^b(\beta) \not\ni \langle v, \rho'\rangle \wedge \rho(x) \neq \rho'(x) \wedge \rho[x \mapsto \rho'(x)] = \rho'(x) \,\}\ .$$

Examples of possible choices for $\mathcal{B}$ include base types of the shape $\beta = \{\nu : t \mid \varphi\}$ where t is a simple type like int and φ a value in a standard relational abstract domain such as octagons and polyhedra that relates ν with the variables in the environment. For instance, when considering the polyhedra domain, basic refinement types can represent values subject to a system of linear constraints, such as the following, where x, y, z are the variables evaluated in the environments:

$$\beta = \{\nu : \mathsf{int} \mid x + y + z \leq \nu \wedge x - y \leq 0 \wedge y + z \leq 2x\}\ .$$

Note that the set notation in the example is just syntactic sugar. The value β is not actually a set, but an element of $\mathcal{B}$ that denotes a set (of pairs) under γ^b.

Dependent effects. Let $\langle \Phi, \sqsubseteq^\phi, \sqcup^\phi, \sqcap^\phi, \bot^\phi, \top^\phi \rangle$ denote a lattice of dependent effects. Similar to basic refinement types, a dependent effect $\phi \in \Phi$ represents a set of pairs $\langle \pi, \rho \rangle$ where π is a trace and ρ captures its environmental dependencies. Again, we formalize this by assuming a monotone and top-strict function $\gamma^\phi \in \Phi \to \wp(\mathcal{V}^* \times Env)$. Similar to basic types, we denote by $\mathsf{dom}(\phi)$ the set of variables that are constrained by ϕ. We assume some additional operations on our abstract domains for dependent types and effects that we will introduce below.

Types. With basic refinement types and dependent effects in place, we define our types as follows:

$$\tau \in \mathcal{T} \ ::= \ \beta \ \mid \ x : (\tau_2 \& \phi_2) \to \tau_1 \& \phi_1 \ \mid \ \exists x : \tau_1 .\, \tau_2 \ .$$

Intuitively, a function type $x : (\tau_2 \& \phi_2) \to \tau_1 \& \phi_1$ describes functions that take an input value x of type τ_2 and a prefix trace described by ϕ_2 such that evaluating the body e produces a result value of type τ_1 and extends the prefix trace to a trace described by ϕ_1. Type refinements in τ_1 may depend on x. Existential types $\exists x : \tau_1 .\, \tau_2$ represent values of type τ_2 that depend on the existence of a witness value x of type τ_1.

We lift the function dom from basic types and effects to types in the expected way:

$$\begin{aligned}
\mathsf{dom}(x : (\tau_2 \& \phi_2) \to \tau_1 \& \phi_1) &= \mathsf{dom}(\tau_2) \cup ((\mathsf{dom}(\phi_2) \cup \mathsf{dom}(\tau_1) \cup \mathsf{dom}(\phi_1)) \setminus \{x\}) \\
\mathsf{dom}(\exists x : \tau_1 .\, \tau_2) &= \mathsf{dom}(\tau_1) \cup (\mathsf{dom}(\tau_2) \setminus \{x\})
\end{aligned}$$

We also lift γ^b to a concretization function $\gamma^{\mathsf{t}} \in \mathcal{T} \to \wp(\mathcal{V} \times Env)$ on types:

$$\begin{aligned}
&\gamma^{\mathsf{t}}(\beta) = \gamma^b(\beta) \\
&\gamma^{\mathsf{t}}(x : (\tau_1 \& \phi_1) \to \tau_2 \& \phi_2) = \mathcal{V} \times Env \\
&\gamma^{\mathsf{t}}(\exists x : \tau_1 .\, \tau_2) = \{ \langle v, \rho \rangle \mid \langle v', \rho \rangle \in \gamma^{\mathsf{t}}(\tau_1) \wedge \langle v, \rho[x \mapsto v'] \rangle \in \gamma^{\mathsf{t}}(\tau_2) \} \ .
\end{aligned}$$

Note that the function γ^{t} uses a coarse approximation of function values. The reason is that we will use γ^{t} to give meaning to typing environments, which we will in turn use to define what it means to strengthen a type with respect to dependencies expressed by a given typing environment. When strengthening with respect to a typing environment, we will only track dependencies to values of base types, but not function types.

We define typing environments Γ as binding lists between variables and types: $\Gamma ::= \varnothing \mid \Gamma, x : \tau$. We extend dom to typing environments as: $\mathsf{dom}(\varnothing) = \emptyset$ and $\mathsf{dom}(\Gamma, x : \tau) = \mathsf{dom}(\Gamma) \cup \{x\}$. We then impose a well-formedness condition $\mathsf{wf}(\Gamma)$ on typing environments. Intuitively, the condition states that bindings in Γ do not constrain variables that are outside of the scope of the preceding bindings in Γ:

$$\textsc{wf-emp}\ \ \mathsf{wf}(\varnothing) \qquad\qquad \textsc{wf-bind}\ \ \frac{\mathsf{wf}(\Gamma) \qquad \mathsf{dom}(\tau) \subseteq \mathsf{dom}(\Gamma) \qquad x \notin \mathsf{dom}(\Gamma)}{\mathsf{wf}(\Gamma, x : \tau)}$$

If $\mathsf{wf}(\Gamma)$ and $x \in \mathsf{dom}(\Gamma)$, then we write $\Gamma(x)$ for the unique type bound to x in Γ.

As previously mentioned, we lift γ^{t} to a concretization function for typing environments:

$$\gamma^{\mathsf{t}}(\varnothing) = Env \qquad \gamma^{\mathsf{t}}(\Gamma, x : \tau) = \gamma^{\mathsf{t}}(\Gamma) \cap \{ \rho \mid \exists v.\, \langle v, \rho \rangle \in \gamma^{\mathsf{t}}(\Gamma(x)) \} \ .$$

Typing judgements. Our type system builds on existing refinement type systems with semantic subtyping [6, 31]. Subtyping judgements take the form $\Gamma \vdash \tau_1 <: \tau_2$ and are defined by the rules in Fig. 3. We implicitly restrict these judgments to well-formed typing environments.

The rule S-BASE handles subtyping on basic types by reducing it to the ordering $\sqsubseteq^b$. Importantly, the basic type β_1 on the left side is strengthened with the environmental dependencies expressed by

$$\text{S-BASE}\ \frac{\beta_1[\Gamma] \sqsubseteq^b \beta_2}{\Gamma \vdash \beta_1 <: \beta_2} \qquad \text{S-WIT}\ \frac{\Gamma \vdash \tau' <: \tau \qquad \Gamma, y : \tau' \vdash \tau_1 <: \tau_2[y/x]}{\Gamma, y : \tau' \vdash \tau_1 <: \exists x : \tau .\, \tau_2} \qquad \text{S-EXISTS}\ \frac{\Gamma, x : \tau \vdash \tau_1 <: \tau_2}{\Gamma \vdash \exists x : \tau .\, \tau_1 <: \tau_2}$$

$$\text{S-FUN}\ \frac{\Gamma \vdash \tau_2' <: \tau_2 \qquad \phi_2'[\Gamma, x : \tau_2'] \sqsubseteq^\phi \phi_2 \qquad \Gamma, x : \tau_2' \vdash \tau_1 <: \tau_1' \qquad \phi_1[\Gamma, x : \tau_2'] \sqsubseteq^\phi \phi_1'}{\Gamma \vdash (x : (\tau_2 \& \phi_2) \to \tau_1 \& \phi_1) <: (x : (\tau_2' \& \phi_2') \to \tau_1' \& \phi_1')}$$

Fig. 3. Semantic subtype relation.

Γ. To this end, we assume the existence of an operator $\beta[\Gamma]$ that satisfies the following specification:

$$\gamma^b(\beta[\Gamma]) \;\supseteq\; \gamma^b(\beta) \cap (\mathcal{V} \times \gamma^{\mathrm{t}}(\Gamma)) \;\;.$$

We require this operator to be monotone in both arguments where $\Gamma \leq \Gamma'$ iff for all $x \in \mathrm{dom}(\Gamma')$, $\Gamma(x) = \Gamma'(x)$. We also assume a strengthening operator $\phi[\Gamma]$ on effects with corresponding assumptions.

The rule S-FUN handles subtyping of function types. As expected, the input type and effect are ordered contravariantly and the output type and effect covariantly. Note that we allow the input effect to depend on the parameter x.

The rule S-EXISTS introduces existential types on the left side of the subtyping relation whereas S-WIT introduces them on the right side. The latter rule relies on an operator $\tau[y/x]$ that expresses substitution of the dependent variable x in type τ by the variable y. This operator is defined by lifting corresponding substitution operators $\beta[y/x]$ on basic types and $\phi[y/x]$ on effects in the expected way. The soundness of these operators is captured by the following assumption:

$$\gamma^b(\beta[y/x]) \supseteq \{\, \langle v, \rho[x \mapsto \rho(y)] \rangle \mid \langle v, \rho \rangle \in \gamma^b(\beta) \,\}$$
$$\gamma^\phi(\phi[y/x]) \supseteq \{\, \langle \pi, \rho[x \mapsto \rho(y)] \rangle \mid \langle \pi, \rho \rangle \in \gamma^\phi(\phi) \,\} \;\;.$$

Typing judgments take the form $\Gamma; \phi \vdash e : \tau \& \phi'$ and are defined by the rules in Fig. 4[1]. Intuitively, such a judgement states that under typing environment Γ, expression e extends the event sequences described by effect ϕ to the event sequences described by effect ϕ' and upon termination, produces a value described by type τ. Again, the typing environments occurring in typing judgements are implicitly restricted to be well-formed. Moreover, we implicitly require $\mathrm{dom}(\phi) \subseteq \mathrm{dom}(\Gamma)$.

The rule T-CONST is used to type primitive values. For this, we assume an operator that maps a primitive value c to a basic type $\{\nu = c\} \in \mathcal{B}$ such that $\gamma^b(\{\nu = c\}) \supseteq \{c\} \times \mathit{Env}$.

The rule T-EV is used to type event expressions **ev** e. For this, we assume an *effect extension operator* $\phi \odot \tau$ that abstracts the extension of the traces represented by effect ϕ with the values represented by the type τ, synchronized on the value environment:

$$\gamma^\phi(\phi \odot \tau) \;\supseteq\; \{\langle \pi \cdot v, \rho \rangle \mid \langle v, \rho \rangle \in \gamma^{\mathrm{t}}(\tau) \wedge \langle \pi, \rho \rangle \in \gamma^\phi(\phi)\} \;\;. \tag{1}$$

We require that $\odot$ is monotone in both of its arguments.

The following is an example judgment for the bid event **ev** 1 expression in the `auction` example from Sec. 2:

$$\Gamma, i : \{\nu \mid \nu >= 1\}; [..q_1 \mapsto \mathit{bids} = i - 1 >= 0] \vdash \textbf{ev}\;\; 1 : \mathrm{unit}\&[..q_1 \mapsto \mathit{bids} = i >= 1]$$

The effect to the left of the turnstile describes event prefixes associated to all the executions leading to the evaluation of expression **ev** 1. It states that q_1 is the only reachable state and the

[1]All auxiliary operators occurring in the typing rules such as $\odot$ are defined below.

$$\textsc{t-const}\quad \Gamma;\phi \vdash c : \{\nu = c\}\&\phi \qquad \textsc{t-var}\quad \Gamma;\phi \vdash x : \Gamma(x)\&\phi \qquad \textsc{t-ev}\ \frac{\Gamma\ ;\ \phi \vdash e : \tau\&\phi'}{\Gamma;\phi \vdash \textbf{ev}\ e : \{\nu = \bullet\}\&(\phi' \odot \tau)}$$

$$\textsc{t-abs}\ \frac{\Gamma, x : \tau_2\ ;\ \phi_2 \vdash e : \tau_1\&\phi_1}{\Gamma\ ;\ \phi \vdash \lambda x.e : (x : \tau_2\&\phi_2 \to \tau_1\&\phi_1)\&\phi}$$

$$\textsc{t-app}\ \frac{\Gamma;\phi \vdash e_1 : \tau_1\&\phi_1 \qquad \Gamma;\phi_1 \vdash e_2 : \tau_2\&\phi_2 \qquad \tau_1 = x : (\tau_2\&\phi_2) \to \tau\&\phi'}{\Gamma;\phi \vdash e_1\ e_2 : \exists x : \tau_2.\,(\tau\&\phi')}$$

$$\textsc{t-weaken}\ \frac{\phi[\Gamma] \sqsubseteq^{\phi} \psi \qquad \Gamma;\psi \vdash e : \tau'\&\psi' \qquad \Gamma \vdash \tau' <: \tau \qquad \psi'[\Gamma] \sqsubseteq^{\phi} \phi'}{\Gamma;\phi \vdash e : \tau\&\phi'}$$

$$\textsc{t-cut}\ \frac{\Gamma;\phi \vdash v : \tau\&\phi \qquad x \notin \mathsf{fv}(e) \qquad \Gamma, x : \tau;\phi \vdash e : \tau'\&\phi'}{\Gamma;\phi \vdash e : \exists x : \tau.\,(\tau'\&\phi')}$$

$$\textsc{t-ite}\ \frac{\Gamma;\phi \vdash x : bool\&\phi_0 \qquad \Gamma[x = \mathsf{true}];\phi_0 \vdash e_1 : \tau\&\phi' \qquad \Gamma[x = \mathsf{false}];\phi_0 \vdash e_2 : \tau\&\phi'}{\Gamma;\phi \vdash \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau\&\phi'}$$

Fig. 4. Typing relation.

accumulator bid is equal to i-1. The typing judgment states that, for all executions, the extended effect that account for a new bidding represented by the observable bid (1) event, preserves the invariant between the accumulator and the program variable i, and that $i >= 1$ according to its type constraints.

The rule T-ITE assumes without loss of generality that only variables are allowed to be used as test conditions. It is defined in terms of an environment strengthening operator $\Gamma[x = c]$ for $x \in \mathsf{dom}(\Gamma)$ defined as $\Gamma[x = c](x) = \Gamma(x) \sqcap^b \{\nu = c\}$ and $\Gamma[x = c](y) = \Gamma(y)$ for $y \in \mathsf{dom}(\Gamma) \setminus \{x\}$.

The notation $\exists x : \tau.\,(\tau'\&\phi)$ used in the conclusion of rules T-APP and T-CUT is a shorthand for $(\exists x : \tau.\,\tau')\&(\exists x : \tau.\,\phi)$, where $\exists x : \tau.\,\phi$ computes the projection of the dependent variable x in effect ϕ, subject to the constraints captured by type τ. That is, this operator must satisfy:

$$\gamma^{\phi}(\exists x : \tau.\,\phi) \supseteq \{\, \langle \pi, \rho[x \mapsto v] \rangle \mid \langle \pi, \rho \rangle \in \gamma^{\phi}(\phi[x : \tau]) \,\}\ .$$

As with our other abstract domain operators, we require this to be monotone in both τ and ϕ.

The rule T-CUT allows one to introduce an existential type $\exists x : \tau.\,\tau'$, provided one can show the existence of a witness value v of type τ' for x. In other dependent refinement type systems, this rule is replaced by a variant of rule S-WIT as part of the rules defining the subtyping relation. We use the alternative formulation to avoid mutual recursion between the subtyping and typing rules.

The remaining rules are as expected. In particular, the rule T-WEAKEN allows one to weaken a typing judgement using the subtyping relation (and ordering on effects), relative to the given typing environment.

Soundness. We prove the following soundness theorem. Intuitively, the theorem states that (1) well-typed programs do not get stuck and (2) the output effect established by the typing judgement approximates the set of event traces that the program's evaluation may generate.

THEOREM 4.1 (SOUNDNESS). *If $\phi \vdash e : \tau\&\phi'$ and $\langle \pi, \rho \rangle \in \gamma^{\phi}(\phi)$, then $\langle e, \pi \rangle \rightsquigarrow \langle e', \pi' \rangle$ implies $e' \in \mathcal{V}$ and $\langle \pi', \rho \rangle \in \gamma^{\phi}(\phi')$.*

The soundness proof details are available in the extended version [46], but we summarize here. The proof of Theorem 4.1 proceeds in two steps. We first show that any derivation of a typing judgement $\phi \vdash e : \tau\&\phi'$ can be replayed in a concretized version of the type system where basic types are drawn from the concrete domain $\wp(\mathcal{V} \times Env)$ and effects from the concrete domain $\wp(\mathcal{V}^* \times Env)$ (i.e., both γ^b and γ^ϕ are the identity on their respective domain). Importantly, in this concretized type system all operations such as strengthening $\tau[\Gamma]$ and effect extension $\phi \odot \tau$ are defined to be precise. That is, we have e.g. $\phi \odot \tau \overset{\text{def}}{=} \{\langle \pi \cdot v, \rho\rangle \mid \langle v, \rho\rangle \in \tau \land \langle \pi, \rho\rangle \in \phi\}$. In a second step, we then show standard progress and preservation properties for the concretized type system.

While one could prove progress and preservation directly for the abstract type system, this would require stronger assumptions on the abstract domain operations. By first lowering the abstract typing derivations to the concrete level, the rather weak assumptions above suffice.

5 Automata-Based Dependent Effects Domain

In this section, we introduce an automata-based dependent effects domain Φ_A that can be used to instantiate the domain of dependent effects Φ assumed by our type and effect system presented in §4. The domain is parametric in an automaton A that specifies the property to be verified for a given program. That is, the dependent effects domain is designed to support solving the following verification problem: given a program, show that the prefixes of the traces generated by the program are disjoint from the language recognized by A. To this end, the abstract domain tracks the reachable states of the automaton: each time the program emits an event, A advances its state according to its transition relation. The set of automata states is in general infinite, so we abstract A's transition relation by abstract interpretation. The abstraction takes into account the program environment at the point where the event is emitted, thus, yielding a domain of *dependent* effects.

5.1 Symbolic Accumulator Automata

Our automaton model is loosely inspired by the various notions of (symbolic) register or memory automata considered in the literature [3, 9, 25]. A *symbolic accumulator automaton (SAA)* is defined over a potentially infinite alphabet and a potentially infinite data domain. In the following, we will fix both of these sets to coincide with the set of primitive values $\mathcal{V}$ of our object language. Formally, an SAA is a tuple $A = \langle Q, \Delta, \langle q_0, a_0\rangle, F\rangle$. We specify the components of the tuple on-the-fly as we define the semantics of the automaton.

A state $\langle q, a\rangle$ of A consists of a control location q drawn from the finite set Q and a value $a \in \mathcal{V}$ that indicates the current value of the accumulator register. The pair $\langle q_0, a_0\rangle$ with $q_0 \in Q$ and $a_0 \in \mathcal{V}$ specifies the initial state of A. The set $F \subseteq Q$ is the set of final control locations.

The symbolic transition relation Δ denotes a set of transitions $\langle q, a\rangle \xrightarrow{v} \langle q', a'\rangle$ that take a state $\langle q, a\rangle$ to a successor state $\langle q', a'\rangle$ under input symbol $v \in \mathcal{V}$. The transitions are specified as a finite set of symbolic transitions $\langle q, g, u, q'\rangle \in \Delta$, written $q \xrightarrow{\{g\}u} q'$, where $g \in \mathcal{G}$ is a *guard* and $u \in \mathcal{U}$ an *(accumulator) update*. Both guards and updates can depend on the input symbol v consumed by the transition and the accumulator value a in the pre state, allowing the automaton to capture non-regular properties and complex program variable dependencies. We make our formalization parametric in the choice of the languages that define the sets $\mathcal{G}$ and $\mathcal{U}$[2]. To this end, we assume denotation functions $[\![g]\!](v, a) \in \mathbb{B}$ and $[\![u]\!](v, a) \in \mathcal{V}$ that evaluate a guard g to its truth value, respectively, an update u to the new accumulator value. We then have $\langle q, a\rangle \xrightarrow{v} \langle q', a'\rangle$ if there exists $q \xrightarrow{\{g\}u} q' \in \Delta$ such that $[\![g]\!](v, a) = \texttt{true}$ and $[\![u]\!](v, a) = a'$. We require that Δ is such that this transition relation is total. For $\pi \in \mathcal{V}^*$, we denote by $\langle q, a\rangle \xrightarrow{\pi}{}^{*} \langle q', a'\rangle$ the reflexive transitive

[2]In our implementation, we use integer arithmetic expressions for $\mathcal{U}$ and conjunctions of (in)equality predicates for $\mathcal{G}$.

closure of this relation and define the *semantics of a state* as the set of traces that reach that state:

$$[\![\langle q,a\rangle]\!] = \{\, \pi \mid \langle q_0, a_0\rangle \xrightarrow{\pi}{}^{*} \langle q,a\rangle \,\} \ .$$

With this, the language of A is defined as

$$\mathcal{L}(A) = \bigcup \{\, [\![\langle q,a\rangle]\!] \mid q \in F \,\} \ .$$

Intuitively, $\mathcal{L}(A)$ is the set of all *bad* prefixes of event traces that the program is supposed to avoid.

5.2 Automata-Based Dependent Effects Domain

We now describe the domain Φ_A. For the remainder of this section, we fix an SAA A and omit subscript A for Φ_A and all its operations.

Concrete automata domain of dependent effects. Recall from §4 that a dependent effect domain Φ represents a sublattice of $\wp(\mathcal{V}^* \times Env)$. Since the states of A represents sets of event traces, a natural first step to define such a sublattice is to pair off automaton states with value environments: $\Phi_C = \wp(Q \times \mathcal{V} \times Env)$.

The corresponding concretization function $\gamma^{\phi}_C : \Phi_C \to \wp(\mathcal{V}^* \times \rho)$ is given by:

$$\gamma^{\phi}_C(\phi_C) = \bigcup_{\langle q,a,\rho\rangle \in \phi_C} \{\, \langle \pi, \rho\rangle \mid \pi \in [\![\langle q,a\rangle]\!] \,\} \ .$$

Since γ^{ϕ}_C is defined element-wise on Φ_C, it is easy to see that it is monotone and preserves arbitrary meets. It is therefore the upper adjoint of a Galois connection between $\wp(\mathcal{V}^* \times Env)$ and Φ_C. Let α^{ϕ}_C be the corresponding lower adjoint, which is uniquely determined by γ^{ϕ}_C.

The operations on the dependent effect domain Φ_C are then obtained calculationally as the best abstractions of their concrete counterparts. In particular, we define:

$$\begin{aligned}\phi_C \odot_C \beta &= \alpha^{\phi}_C(\{\, \langle \pi \cdot v, \rho\rangle \mid \langle v,\rho\rangle \in \gamma^b(\beta) \wedge \langle \pi,\rho\rangle \in \gamma^{\phi}_C(\phi_C) \,\})\\ &= \{\, \langle q', a', \rho\rangle \mid \exists \langle v,\rho\rangle \in \gamma^b(\beta), \langle q,a,\rho\rangle \in S.\, \langle q,a\rangle \xrightarrow{v} \langle q',a'\rangle \,\} \ .\end{aligned}$$

The characterization of $\odot_C$ relies on the fact that the transition relation of the automaton is total. Note that the soundness condition on $\odot_C$ imposed in §4 is obtained by construction from the properties of Galois connections:

$$\gamma^{\phi}_C(\phi_C \odot_C \beta) \supseteq \{\, \langle \pi \cdot v, \rho\rangle \mid \langle v,\rho\rangle \in \gamma^b(\beta) \wedge \langle \pi,\rho\rangle \in \gamma^{\phi}_C(\phi_C) \,\} \ .$$

The remaining operations $\phi[\Gamma]$, $\phi_C[y/x]$, and $\exists x : \tau.\, \phi_C$ are obtained accordingly.

Abstract automata domain of dependent effects. Since the elements $S \in \Phi_C$ can be infinite sets, the operations on Φ_C such as $\odot_C$ are typically not computable. We therefore layer further abstractions on top of Φ_C to obtain an abstract automata domain of dependent effects with computable operations.

We proceed in two steps. Firstly, we change the representation of our abstract domain elements by partitioning the elements of each $\phi_C \in \Phi_C$ based on the control location of the automaton state. That is, we switch to the effect domain $\Phi_R = Q \to \wp(\mathcal{V} \times Env)$, ordered by pointwise subset inclusion. The corresponding concretization function $\gamma^{\phi}_R \in \Phi_R \to \Phi_C$ is given by

$$\gamma^{\phi}_R(\phi_R) = \{\, \langle q,a,\rho\rangle \mid \langle a,\rho\rangle \in \phi_R(q) \,\} \ .$$

Clearly, we do not lose precision when changing the representation of the elements $\phi_C \in \Phi_C$ to elements of Φ_R. In fact, γ^{ϕ}_R is a lattice isomorphism. Its inverse $\alpha^{\phi}_R = {\gamma^{\phi}_R}^{-1}$ is the lower adjoint of a Galois connection between Φ_C and Φ_R.

As before, we obtain the abstract domain operations on Φ_R by defining them as the best abstractions of their counterparts on Φ_C. In particular, we define

$$\phi_R \odot_R \beta = \alpha_R^\phi(\gamma_R^\phi(\phi_R) \odot_C \beta) = \lambda q'.\, \{\, \langle a', \rho \rangle \mid \exists q', \langle v, \rho \rangle \in \gamma^b(\beta), \langle a, \rho \rangle \in \phi_R(q).\, \langle q, a \rangle \xrightarrow{v} \langle q', a' \rangle \,\} \ .$$

Now consider again our abstract domain of base refinement types $\langle \mathcal{B}, \sqsubseteq^b, \bot^b, \top^b, \sqcup^b, \sqcap^b \rangle$ that we have assumed as a parameter of the type and effects system of §4. Recall that each element $\beta \in \mathcal{B}$ abstracts a relation between values and value environments: $\gamma^b(\beta) \subseteq \wp(\mathcal{V} \times Env)$. We can thus reuse this domain to abstract the relations $\phi_R(q) \subseteq \wp(\mathcal{V} \times Env)$ between the reachable accumulator values at location q of the automaton and the environments. This leads to the following definition of our final automata-based effect domain: $\Phi = Q \to \mathcal{B}$. The accompanying concretization function $\gamma_\mathcal{B}^\phi \in \Phi \to \Phi_R$ is naturally obtained by pointwise lifting of γ^b: $\gamma_\mathcal{B}^\phi(\phi) = \gamma^b \circ \phi$. The overall concretization function $\gamma^\phi : \Phi \to \wp(\mathcal{V}^* \times Env)$ is defined by composition of the intermediate concretization functions: $\gamma^\phi = \gamma_C^\phi \circ \gamma_R^\phi \circ \gamma_\mathcal{B}^\phi$.

We then define the operations on Φ in terms of the operations on $\mathcal{B}$. Again, we focus on the operator $\odot$. The remaining operations are defined similarly.

Our goal is to ensure that the overall soundness condition on $\odot$ is satisfied. We achieve this by defining $\phi \odot \beta$ such that

$$\gamma_\mathcal{B}^\phi(\phi \odot \beta) \ \supseteq\ \gamma_\mathcal{B}^\phi(\phi) \odot_R \beta \ . \tag{2}$$

Assuming (2) the overall soundness of $\odot$ then follows by construction:

Lemma 5.1. *For all $\phi \in \Phi$ and $\beta \in \mathcal{B}$, $\gamma^\phi(\phi \odot \beta) \supseteq \{\langle \pi \cdot v, \rho \rangle \mid \langle v, \rho \rangle \in \gamma^t(\beta) \wedge \langle \pi, \rho \rangle \in \gamma^\phi(\phi)\}$.*

Let us thus define an appropriate $\odot$ that satisfies (2). To this end, we first expand $\gamma_\mathcal{B}^\phi(\phi) \odot_R \beta$:

$$\begin{aligned}
&\gamma_\mathcal{B}^\phi(\phi) \odot_R \beta \\
&= \lambda q'.\, \{\, \langle a', \rho \rangle \mid \exists q', \langle v, \rho \rangle \in \gamma^b(\beta), \langle a, \rho \rangle \in \gamma^b(\phi(q)).\, \langle q, a \rangle \xrightarrow{v} \langle q', a' \rangle \,\} \\
&= \lambda q'.\, \bigcup_{q \xrightarrow{\{g\}u} q' \in \Delta} \{\, \langle a', \rho \rangle \mid \exists \langle v, \rho \rangle \in \gamma^b(\beta), \langle a, \rho \rangle \in \gamma^b(\phi(q)).\, [\![g]\!](v, a) = \mathtt{true} \wedge [\![u]\!](v, a) = a' \,\} \ .
\end{aligned}$$

The last equation suggests that we can compute $\phi \odot \beta$ by abstracting for each q', each symbolic transition $q \xrightarrow{\{g\}u} q' \in \Delta$ of the automaton separately, and then take the join of the results. In order to abstract a symbolic transition, we need appropriate abstractions of the semantics of guards and updates with respect to base refinement types. For the sake of our formalization, we therefore assume an abstract interpreter $[\![\cdot]\!]^\# : (\mathcal{G} \cup \mathcal{U}) \to \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ such that for all $t \in \mathcal{G} \cup \mathcal{U}$ and $\beta, \beta' \in \mathcal{B}$

$$\gamma^b([\![t]\!]^\#(\beta, \beta')) \supseteq \{\, \langle v', \rho \rangle \mid \exists v, a.\, \langle v, \rho \rangle \in \gamma^b(\beta) \wedge \langle a, \rho \rangle \in \gamma^b(\beta') \wedge v' = [\![t]\!](v, a) \,\} \ .$$

We then derive $\phi \odot \beta$ from the above equation as follows:

$$\phi \odot \beta = \lambda q'.\, \bigsqcup_{q \xrightarrow{\{g\}u} q' \in \Delta} \{\, [\![u]\!]^\#(\beta \sqcap \beta_g, \phi(q) \sqcap \beta_g) \mid \beta_g = (\exists \nu.\, [\![g]\!]^\#(\beta, \phi(q)) \sqcap \{\nu = \mathtt{true}\}) \,\} \ .$$

Here, β_g captures the environments ρ shared by β and $\phi(q)$ for which the guard g evaluates to `true`. It is used to strengthen β and $\phi(q)$ when evaluating the update expression u. We here assume that $\mathcal{B}$ provides an operator $\exists v.\beta$ that projects out the value component of the pairs represented by some $\beta \in \mathcal{B}$. That is, we must have:

$$\gamma^b(\exists v.\, \beta) = \{\, \langle v, \rho \rangle \mid \exists v'.\, \langle v', \rho \rangle \in \gamma^b(\beta) \,\} \ .$$

The fact that $\odot$ indeed satisfies (2) then follows from the assumption on the abstract interpreter for guards and expressions as well as the soundness of the abstract domain operations of $\mathcal{B}$.

Lemma 5.2. *For all $\phi \in \Phi$ and $\beta \in \mathcal{B}$, $\gamma^{\phi}_{\mathcal{B}}(\phi \odot \beta) \supseteq \gamma^{\phi}_{\mathcal{B}}(\phi) \odot_R \beta$.*

Similarly, monotonicity of $\odot$ follows immediately from the monotonicity of the operations on $\mathcal{B}$.

The remaining operators on Φ assumed in §4 (i.e., $\phi[\Gamma]$, $\phi[y/x]$, and $\exists x : \tau.\, \phi$) are obtained directly by a pointwise lifting of the corresponding operators on $\mathcal{B}$.

6 Automated Inference and Verification

We now describe how to verify temporal safety properties of higher order programs, through automatic inference of accumulative types and effects. To facilitate the calculation of precise effects we build upon the existing data flow refinement type inference algorithm [49] based on abstract interpretation. We provide an abridged description of the original algorithm and explain how we adapt it for our purposes. We then briefly discuss the soundness of the resulting algorithm and how it can be used to automatically verify temporal safety properties.

6.1 Type and Effect Inference by Abstract Interpretation

The data flow type inference, as described by [49], employs a calculational approach in an abstract interpretation style to iteratively compute a dependent refinement type for every subexpression of a program. The corresponding inference algorithm is implemented in the Drift tool. The algorithm is parametric in the choice of an abstract domain of basic types $\mathcal{B}$ (which coincides with our parametrization of the types and effect system) as well as the supported primitive operations on values represented by these basic types (e.g., arithmetic operations, etc.).

The Drift algorithm is a whole program analysis. It assumes that every subexpression e of the program is labeled with a unique program location ℓ, written e_ℓ. The abstract domain consists of *execution maps* $M^\# \in \mathcal{M}^\#$. Roughly speaking, an execution map assigns a type to every program location ℓ. The type inference works by iteratively computing a fixpoint of an abstract transformer $\mathrm{step}^\#[\![-]\!]$. The abstract transformer takes an expression e_ℓ and an execution map $M^\#$, and computes an updated execution map reflecting the data flow in e_ℓ based on what values may occur at each program location as specified by $M^\#$. The abstract transformer is defined by structural recursion over e_ℓ.

At a conceptual level, we simply instantiate the Drift algorithm by treating event sequences as values that can be manipulated directly by the program, akin to the translation approach. The only primitive operator defined on event sequences is $e_1 \cdot e_2$ where e_1 is expected to evaluate to an event sequence π_1 and e_2 to a value v_2. The result of the operation is the concatenated event sequence $\pi_1 \cdot v_2$. We additionally have the constant expression ϵ denoting the empty event sequence. We also have a built-in pair constructor $\langle e_1, e_2 \rangle$ and projection operators $\#_1(e)$ and $\#_2(e)$ on pairs. Event sequences are then abstracted by treating an abstract effect ϕ as yet another kind of base type.

However, instead of just applying the instantiated Drift algorithm on translated programs that manipulate pairs of values and event sequences, we specialize the abstract transformer to take advantage of the knowledge that every expression e_ℓ evaluates to such a pair. As such it can fuse together what would otherwise be costly joins and projections needed for analysis of the product construction. Moreover, the specialized abstract transformer interprets the sequence concatenation operator using the abstract effect domain. This is in contrast to a translation approach where, say, for the SAA effect domain, we would embed the automatons transfer function into the analyzed program and abstract it using the ordinary base types in $\mathcal{B}$. This specialization is key to improving both the efficiency and precision of the resulting analysis.

$$
\begin{array}{l|l}
(x : \tau_{1i} \to \tau_{1o}) \ltimes (x : \tau_{2i} \to \tau_{2o}) \overset{\text{def}}{=} & \tau_1 \ltimes \tau_2 \overset{\text{def}}{=} \langle \tau_1, \tau_1 \sqcup \tau_2 \rangle \\
\quad \textbf{let}\ \langle \tau'_{2i}, \tau'_{1i} \rangle = \tau_{2i} \ltimes \tau_{1i}\ \textbf{in} & \langle \tau_{1a}, \tau_{1b} \rangle \ltimes \langle \tau_{2a}, \tau_{2b} \rangle \overset{\text{def}}{=} \\
\quad \textbf{let}\ \langle \tau'_{1o}, \tau'_{2o} \rangle = \tau_{1o}[x : \tau_{2i}] \ltimes \tau_{2o}[x : \tau_{2i}]\ \textbf{in} & \quad \langle \langle \tau_{1a}, \tau_{1b} \rangle, \langle \tau_{1a} \sqcup \tau_{2a}, \tau_{1b} \sqcup \tau_{2b} \rangle \rangle \\
\quad \langle x : \tau'_{1i} \to \tau'_{1o}\ ,\ x : \tau'_{2i} \to \tau'_{2o} \rangle & \tau_1 \ltimes \top^b \overset{\text{def}}{=} \langle \top^b, \top^b \rangle \\
(x : \tau_1 \to \tau_2) \ltimes \bot^b \overset{\text{def}}{=} \langle x : \tau_1 \to \tau_2\ ,\ x : \bot^b \to \bot^b \rangle &
\end{array}
$$

Fig. 5. Data flow propagation

To build more intuition, we describe the specialized abstract transformer in some more detail. Its precise signature is

$$\mathsf{step}^{\#}[\![-]\!] : \mathcal{E} \to (Env^{\#} \times \Phi) \to \mathcal{M}^{\#} \to ((\mathcal{T} \times \Phi) \times \mathcal{M}^{\#})\ .$$

Intuitively, for each well-formed expression e_ℓ in a given environment $\Gamma \in Env^{\#}$ and effect context $\phi \in \Phi$, and for a given execution map $M \in \mathcal{M}$, the transformer $\mathsf{step}^{\#}[\![e_\ell]\!](\Gamma, \phi)(M)$ returns the updated abstract value and effect at ℓ, along with an updated execution map.

At the core of the definition of $\mathsf{step}^{\#}[\![-]\!]$ lies the monotonic data flow propagation function $\tau_1 \ltimes \tau_2$ on refinement types shown in Fig. 5. Intuitively, it ensures that an argument type τ at the call site of a function f is propagated back to f's definition site. After inferring the result type τ' of f for τ from f's body, τ' is in turn propagated forward to f's call site. For example, if τ_1 is the current inferred type of some variable x bound at location ℓ_1, and τ_2 is the current type inferred for some usage of x at location ℓ_2, then $\tau_1 \ltimes \tau_2$ returns a new pair of types $\langle \tau'_1, \tau'_2 \rangle$ for locations ℓ_1 and ℓ_2 that reflects the forward data flow from ℓ_1 to ℓ_2 and backward data flow from ℓ_2 to ℓ_1.

In most cases, the abstract transformer behaves according to the original definition in the Drift algorithm and, additionally, simply carries along the effect. The most interesting case is for event emission **ev** e_1, shown on the right, which we discuss in more detail. The definition uses a similar monadic style as in [49] that treats $\mathsf{step}^{\#}[\![-]\!]$ as a state monad over execution maps. We use **do** notation for the monadic composition, allowing **let** to introduce new bindings, and we assume two operations, **get** and **update**, that read from or write to the execution map encapsulated by the monad. The transformer starts by extracting the type and effect currently associated with ℓ from the execution map and takes a recursive step on the expression e_i that computes the value to be emitted. The **assert**() construct aborts with the current execution map if the type inferred from e_i is still $\bot^b$ (indicating that e_i has not yet produced an abstract value in the current iteration of the abstract interpretation). Otherwise, it continues by computing the abstract result of the event emission using the effect extension operator $\odot$ of the abstract effect domain. It then uses data flow propagation with the old type and effect at ℓ to compute the new $\langle \tau'_\ell, \phi'_\ell \rangle$. This pair is then written back to the execution map at ℓ and returned.

```
$\mathsf{step}^{\#}[\![(\textbf{ev}\ e_1)_\ell]\!](\Gamma, \phi) \triangleq \textbf{do}$
  $\langle \tau_\ell, \phi_\ell \rangle \leftarrow \textbf{get}\ (\Gamma, \ell)$
  $\langle \tau_1, \phi_1 \rangle \leftarrow \mathsf{step}^{\#}[\![e_1]\!](\Gamma, \phi)$
  $\textbf{assert}(\tau_1 \neq \bot)$
  $\textbf{let}\ \_, \langle \tau'_\ell, \phi'_\ell \rangle = \langle \{\nu = \bullet\}[\Gamma], \phi \odot \tau_1 \rangle \ltimes \langle \tau_\ell, \phi_\ell \rangle$
  $\textbf{update}(\ell, \langle \tau'_\ell, \phi'_\ell \rangle)$
  $\textbf{return}\ \langle \tau'_\ell, \phi'_\ell \rangle$
```

6.2 Soundness of Type and Effect Inference

A challenge in connecting the type inference result with our type and effects system is that the inference algorithm has been proven sound with respect to a bespoke dataflow semantics of functional program rather than a standard operational semantics like the one underlying our system. However, [49] shows that the inference result yields a valid typing derivation in a bespoke data flow refinement type system. To bridge the gap in the soundness argument, we relate the Drift type system with our type and effect system at the abstract level by showing that, from the typing derivation for a translated program produced by the soundness proof of [49], one can reconstruct a typing derivation in the types and effects system for the original effectful program. Further details can be found in the extended version [46] of this paper.

6.3 Automated Verification

As discussed in Sec. 2, our abstract effect domain seeks to improve over a direct approach of translating an input program/property of effects into an effect-free product program that carries its effect trace and employs existing assertion checking techniques [30, 33, 49, 67]. This algorithm places a substantial burden on the type system (or other verification strategy) to track effect sequences as program values that flow from each (translated) event expression to the next. In this strategy, where an **ev** e expression occurred in the original input program, the translated program has an event prefix variable (and accumulator variable) and constructs an extended event sequence. Unfortunately, today's higher-order program verifiers do not have good methods for summarizing program value sequences, nor do they exploit the automaton structure to organize possible sequence values. Thus, those tools struggle to validate the later **assert**ions.

The inference discussed above offers an alternative verification algorithm. Once effects are inferred through the instantiation of our effect abstract domain (over the translated event sequences), it is straightforward to construct a verification algorithm. One merely has to ensure that at every program location ⓘ, the computed summary associates $\bot$ with every accepting state $q_{err}^{ⓘ}$. Our organization of event prefixes around concrete automaton states allows us to better summarize those prefixes into categories, and can be thought of as a control-state-wise disjunctive partitioning. Thus, at each **ev** e expression, the (dataflow) type system directly updates each q's summary with the next event. Sec. 8 experimentally evaluates both of these algorithms and compares them.

7 Implementation, Trace Partitioning, and Benchmarks

Implementation. We implemented both the tuple/CPS translation (Sec. 2) and the type and effect inference (Sec. 6) verification algorithms in a prototype tool called **ev**Drift, as an extension to the Drift [49] type inference tool, which builds on top of the Apron library [23] to support various numerical abstract domains of type refinements.

evDrift takes programs written in a subset of OCaml along with an automaton property specification file as input. **ev**Drift supports higher-order recursive functions, operations on primitive types such as integers and booleans, as well as a non-deterministic if-then-else branching operator. The property specification lists the set of automaton states, a deterministic transition function and an initial state. The specification also includes two kinds of effect-related assertions: those that must hold after every transition, and those that must hold after the final transition. Assertions related to program variables (as in Drift) may be specified in the program itself. Whereas assertions related to effects may be specified in the property specification file.

We also implemented three improvements to the dataflow abstract interpretation. First, we integrate Apron's grid-polyhedra abstract domain [10]—a reduced product of the polyhedra [7] and the grid [1] abstract domains—to interpret type refinements of the form of $x \equiv y \mod 2$. Second, we implemented trace partitioning [51] for increased disjunctive precision. Although these benefits are somewhat orthogonal to our contributions, our evaluation (Sec. 8) also experimentally quantifies the disjunctive benefit of trace partitioning in our setting vis-a-vis the benefit of our

```
let rec order d c =
  if d > 0 then
    if d mod 2 = 0
    then ev c
    else ev (-c);
    order (d - 2) c
  else 0
let _ (dd cc:int) =
  order dd cc
```

Only "c" or "-c" events

```
let rec spend n =
  ev (-1);
  if n <= 0 then 0
  else spend (n-1)
let _ (gas n:int) =
  if gas >= n and
    n >= 0
  then
    (ev gas; spend n)
  else 0
```

$\sum_i^N \leq$ gas

```
let rec reent d =
  ev 1; (* Acq *)
  if d > 0 then
    if nondet() then
      reent (d-1);
      ev -1 (* Rel *)
    else skip
let _ d =
  reent d;
  ev -1 (* Rel *)
```

Rel ≤ # Acq

```
let rec compute vv bound inc =
  ev vv;
  if vv = bound then 0 else
    compute (inc vv) bound inc
let min_max v =
  let f = (fun t ->
    if v>=0 then t-1 else t+1) in
  if v>=0
  then compute v (-1 * v) f
  else compute v (-1 * v) f
```

$\forall i > 0. -v < \pi[i] < v$

Fig. 6. Further examples of our benchmarks. (See the supplement for sources and automata specifications.)

abstract effect domain. Third, we optimize the propagation step of the analysis by implementing the suggestion in [49]. (Note that these three improvements also benefit the prior Drift tool.)

Trace Partitioning. To improve the precision in our analysis, we implemented a type inference algorithm with trace partitioning [51], but instantiating it here in a higher-order setting. Consider the example to the right adapted from [51]. It is easy to see that this program does not raise an assertion error as z is either equal to 1 or -1. However, when using convex abstract domains like polyhedra and octagons, z will have an abstract representation that includes the integer 0 because the abstract domain elements of the two branches of the conditional are joined together. Consequently, an analysis would raise an undesirable assertion error.

```
let f x y =
  let z =
    if y >= 0 then 1
    else -1
  in
  assert z != 0; x/z
```

Roughly speaking, with trace-partitioning, every if-then-else expression is analyzed twice: once in a context where the condition is true and once for false. This directly alleviates the problem described in the above example as `z≠0` in the abstract representation of either of the branches. Beyond if-then-else expressions, trace-partitioning is also useful to increase precision in the analysis of functions with multiple callsites. As a demonstration, consider the example on the right. Using convex abstract domains, the argument `y1` to the function `g` would have an abstract representation that includes the integer 0. In the same spirit as for if-then-else expressions, we analyze the function `g` in separate contexts for each call site.

```
let g x1 y1 =
  assert y1 != 0; x1/y1
let f x2 y2 =
  g x2 y2 + g (-x2) y2
```

The original Drift type system [49] extends the typing judgement by including call stacks to infer distinct refinement types for program nodes under distinct call stacks. The extension for if-then-else partitioning, which involves program traces composed of locations where the conditional branching takes different paths, largely works in a similar way. The only difference is that program traces, along with the respective context of the chosen condition, are *emitted* by program nodes which have if-then-else expressions and are implemented as lists, whereas call stacks are naturally implemented as stacks. Importantly, for this work, these extensions to the Drift type system can easily carry over to the **ev**Drift type system. We give more details about this extended type system in the extended version [46] of this paper.

Benchmarks. To our knowledge there are no existing benchmarks for higher-order programs with the general class of SAA properties described, although there are related examples in some fragments of SAA. We thus created such a suite from the literature, extending them, and creating new ones. We plan to contribute these 23 benchmarks to SV-COMP [4]. Figure 6 lists some of them.

These benchmarks test our tool to verify a variety of SAA properties like (left-to-right in Fig. 6) tracking disjoint branches of a program, resource analysis, verifying a reentrant lock, and tracking the minimum/maximum of a program variable. Other examples use the accumulator for summation, maximum/minimum, monotonicity, etc. similar to those found in automata literature [3, 9, 25]. We also include an auction smart contract [59] and adapt some example programs proposed in [44]. These benchmarks involve verification of amortized analysis [19, 20, 22] for a pair of queues, and the verification of liveness and fairness for a non-terminating web-server. Finally, for several benchmarks, we created corresponding *unsafe* variants by tweaking the program or property. All benchmarks are provided in the supplement, and publicly available (*URL omitted for reviewing*).

8 Evaluation

We sought to answer two research questions:

(1) How does **ev**Drift compare with other state-of-the-art automated verification tools for higher-order programs?

(2) What is the effect of trace-partitioning on efficiency and accuracy?

Comparing our approach with other methods. We aim to compare against the somewhat mature prior tools Drift, RCaml/Spacer, MoCHi [53] and ReTHFL [28] which support higher-order programs and can validate assertions and even some temporal properties. Murase et al. [43] focus on liveness properties and reduce the problem to termination (also see Secs. 1 and 9 for discussions of other works and tools). Drift, RCaml/Spacer, MoCHi, and ReTHFL do not directly operate on programs with effects, so we used our reduction to verifying assertions of higher-order (effect-free) programs, which also enables those tools to technically now be applied to SAA properties. We first apply a selective store passing transformation to effectful programs (based on an algorithm adapted from Nielsen [48]), producing an optimized product program that preserves parts of the original effectful program. The translation is guided by whether an expression observes an event generated during its evaluation. Consequently, the configuration is passed selectively to only those expressions that may observe such events. Drift is discussed in Sec. 6. RCaml/Spacer[3] is based on extensions of Constrained Horn Clauses, is part of CoAR [63], and is built on top of several prior works [30, 37, 54]. MoCHi [33] is a CEGAR-style software model checker based on higher-order recursion schemes and relies on either interpolating theorem provers or an ICE-based solver of Constrained Horn Clauses (CHC) for predicate discovery. ReTHFL is a type-based validity checker for a fragment νHFL(Z) of HFL(Z), a higher-order fixed point logic extended with integers, to which the verification of higher-order functional programs is known to be reducible, and it leverages CHC solvers to infer predicates within a bespoke refinement type system for νHFL(Z). We also considered LiquidHaskell [65], which includes an implementation [24]. However, LiquidHaskell is somewhat incomparable because (i) it requires user interaction whereas our aim is full automation and (ii) the eager-versus-lazy evaluation order difference impacts the language semantics and possible event traces, so it is difficult to perform a meaningful comparison. For each tool, we use the latest version available at the time of experiments and corresponded with the respective developers to ensure proper usage. All our experiments were conducted on an x86_64 AMD EPYC 7452 32-Core Machine with 125 Gi memory. We used BenchExec 3.29 [58] to ensure precise measurement for each run.

In our **ev**Drift experiments, we run all our benchmarks using several configurations: various combinations of context sensitivity, trace partitioning, numerical abstract domains, and added precision for inference of effect sequences. Context sensitivity—denoted “*cs*”—is either set to "none" (0) or else to a call-site depth of 1. So for example, a configuration with context-sensitivity 1 and

[3]We use RCaml/Spacer at commit `299e979bfce7d9b0532586bfc42b449fd0451531` with the CoAR config `config/solver/rcaml_wopp_spacer.json`.

trace partitioning enabled remembers the last call site and also the last if-else branch location. For research question #1, we evaluate the end-to-end improvement of all of our work on **ev**Drift over existing tools, so we use Drift (plus the tuple translation) *without* trace partitioning—denoted "tp:F"—and **ev**Drift *with* trace partitioning "tp:T". Further below in research question #2 we evaluate the degree to which **ev**Drift improves the state of the art due to the use of the abstract effect domain, versus through the use of trace partitioning (as well as the performance overhead of trace partitioning). Regarding abstract domains, we use the loose version of the polyhedra domain [49] for all our benchmarks except for those that involve mod operations where we use the grid-polyhedra domain. For polyhedra, we further consider two different widening configurations: standard widening and widening with thresholds. For widening with thresholds [5]—denoted "*th*"—we use a simple heuristic that chooses the conditional expressions in the analyzed programs as well as pairwise inequalities between the variables in scope as constraints. The grid-polyhedra domain does not properly support threshold widening, so we only use standard widening here. Finally, we add an option—denoted "*io*"—to increase precision for effect sequences through relationships between accumulator variables before and after a function body is evaluated. *io* helps capture the consequence every function has over accumulator variables more precisely, albeit at the cost of some performance due to increased number of variables in the abstract environment. In the discussion below, we report only the result for the configuration that verified the respective benchmark in the least amount of time (as identified in **Config** columns in the tables). For instances where all versions fail to verify a benchmark, we report results for the fastest configuration for brevity. We also include results for other configurations in the extended version [46] of this paper.

Table 1 summarizes the results of our comparison. **ev**Drift significantly outperforms the other three tools in terms of number of benchmarks verified and efficiency. Drift via tuple reduction was only able to verify 12 of the 23 benchmarks, while **ev**Drift could verify 21. Moreover, **ev**Drift could also verify the `market` benchmark with configuration ⟨*cs*:2, *tp*:*F*, *io*:*T*, *ls*⟩ in under 30 seconds. Across all benchmarks that Drift could solve, it had a geomean of 1.9 s. Across all benchmarks that **ev**Drift could solve, it had a geomean of 0.9 s. For those benchmarks that *both* Drift and **ev**Drift could verify, **ev**Drift was 6.3× faster. RCaml/Spacer successfully verified 6 out of 23 benchmarks, with a geomean of 1.6 s across these, and was unable to verify the others due to either imprecision, timeout, or memory blowup. ReTHFL verified 18 benchmarks for which it reported a geomean of 5.1 s. Using the web interface for ReTHFL [29], we found that an additional benchmark (`nondet_max`) is verified in under 5 s, suggesting that configuration issues in the running environments caused the difference. Moreover, the verification of `market` and `lics18-amortized` using the web interface was not conclusive, with the printed output remaining in a seemingly frozen state. Finally, MoCHi verified 11 benchmarks for which it reported a geomean of 6.7 s. As anticipated, the cross-product transformation of the original program and property significantly increases the program size, thus requiring high context-sensitivity, which no existing tool provides with high precision. In addition, we also ran **ev**Drift on the unsafe variants of our benchmarks. We used the same configurations as for their respective safe versions in Table 1. The individual results are omitted for lack of space, but **ev**Drift analyzed all unsafe benchmarks in 273 seconds, returning unknown on each of them. (The tool can only prove the absence of errors, not their presence.)

We deduced at least three major factors behind **ev**Drift's superior performance. (1) **ev**Drift evaluates the **ev** expressions inline which reduces program size significantly. This leads to significantly faster runtimes and smaller memory footprint for **ev**Drift for all benchmarks. This also reduces a function call and the need to remember another call site for **ev**Drift in some cases like `overview1` and `sum-appendix` where the **ev** expression might have different arguments at different locations. Moreover, some benchmarks require the inference of non-convex input-output relations for functions, which the used numerical abstract domains cannot express. This is why **ev**Drift can

Table 1. Comparison of **ev**Drift against assertion verifiers for effect-free programs: Drift, RCaml/Spacer, MoCHi and ReTHFL via our tuple reduction. For each verifier, we show the result ("✔" for successful verification; "?" for unknown; "☢" for some failure other than unknown; "**T**" for timeout - over 900 seconds; "**M**" for out of memory - over 2GB), CPU time in seconds, maximum memory used in megabytes and the chosen configuration for Drift and **ev**Drift. For Drift and **ev**Drift, the configuration tuples show what context-sensitivity (*cs*) was used, if trace-partitioning was used (*tp*), if added precision for effects was used (*io*), and which abstract domain was used (*ls* for loose-polyhedra, and *pg* for grid-polyhedra). **evDrift verified additional 9**, **15**, **10** and **3** that Drift, RCaml/Spacer, MoCHi and ReTHFL, respectively, could not, and it is **6.3× faster** than Drift on Drift-verifiable examples, **5.3× faster** than RCaml/Spacer on RCaml/Spacer-verifiable examples, **16.8× faster** than MoCHi on MoCHi-verifiable examples, and **6.4× faster** than ReTHFL on ReTHFL-verifiable examples.

	Prior Tools (via Tuple Reduction)											
	Drift			RCaml		MoCHi		ReTHFL		**ev**Drift		
Bench	**Res**	**CPU**	**Config.**	**Res**	**CPU**	**Res**	**CPU**	**Res**	**CPU**	**Res**	**CPU**	**Config.**
1. `all-ev-pos`	✔	0.6	$\langle cs{:}0, ls\rangle$	✔	0.8	✔	0.8	✔	2.5	✔	0.2	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
2. `alt-inev`	?	60.8	$\langle cs{:}0, pg\rangle$	**T**	901.1	?	69.2	✔	4.7	✔	2.4	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
3. `auction`	?	55.1	$\langle cs{:}1, ls\rangle$	**T**	901.1	?	90.7	✔	18.6	✔	2.7	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
4. `binomial_heap`	?	544.1	$\langle cs{:}1, pg\rangle$	**T**	901.1	**M**	581.2	✔	4.3	✔	2.2	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
5. `concurrent_sum`	✔	1.7	$\langle cs{:}0, ls\rangle$	**M**	12.4	?	190.7	✔	4.2	✔	0.2	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
6. `depend`	✔	0.1	$\langle cs{:}0, ls\rangle$	✔	0.1	✔	0.7	✔	1.9	✔	0.0	$\langle cs{:}1, tp{:}T, io{:}F, ls\rangle$
7. `disj-gte`	?	35.8	$\langle cs{:}0, pg\rangle$	**M**	505.6	✔	198.6	✔	4.9	✔	2.2	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
8. `disj-nondet`	?	10.1	$\langle cs{:}0, ls\rangle$	**M**	528.5	✔	45.0	✔	3.8	✔	2.3	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
9. `higher-order`	✔	1.5	$\langle cs{:}0, ls\rangle$	✔	13.2	✔	16.4	✔	3.4	✔	0.6	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
10. `intro-ord3`	✔	24.0	$\langle cs{:}1, ls\rangle$	**M**	164.6	?	64.1	?	8.1	✔	3.8	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
11. `lics18-amortized`	?	273.6	$\langle cs{:}0, pg\rangle$	**M**	307.6	**M**	228.8	**T**	901.1	✔	6.4	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
12. `lics18-hoshrink`	?	9.9	$\langle cs{:}0, pg\rangle$	?	0.1	?	1.3	?	3.7	?	7.0	$\langle cs{:}1, tp{:}F, io{:}F, pg\rangle$
13. `lics18-web`	?	55.0	$\langle cs{:}0, ls\rangle$	☢	3.7	**T**	901.2	✔	11.2	✔	7.1	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
14. `market`	?	276.9	$\langle cs{:}1, ls\rangle$	**M**	481.9	**T**	901.1	**M**	5.5	?	56.2	$\langle cs{:}0, tp{:}F, io{:}T, pg\rangle$
15. `max-min`	?	143.3	$\langle cs{:}1, pg\rangle$	**M**	28.3	**T**	900.9	✔	78.5	✔	43.1	$\langle cs{:}1, tp{:}T, io{:}T, ls\rangle$
16. `monotonic`	✔	2.3	$\langle cs{:}0, ls\rangle$	**T**	901.1	✔	2.8	✔	4.1	✔	0.5	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
17. `nondet_max`	✔	2.3	$\langle cs{:}0, ls\rangle$	✔	1.8	**T**	901.1	**T**	901.0	✔	0.6	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
18. `num_evens`	✔	9.2	$\langle cs{:}0, ls\rangle$	**T**	900.7	✔	17.2	✔	4.3	✔	4.7	$\langle cs{:}1, tp{:}F, io{:}T, ls\rangle$
19. `order-irrel-nondet`	?	26.2	$\langle cs{:}1, pg\rangle$	✔	2.9	✔	61.9	✔	8.6	✔	2.7	$\langle cs{:}1, tp{:}T, io{:}T, ls\rangle$
20. `overview1`	✔	1.8	$\langle cs{:}1, ls\rangle$	✔	1.7	✔	3.5	✔	2.5	✔	0.3	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
21. `reentr`	✔	3.4	$\langle cs{:}0, ls\rangle$	**T**	900.1	?	590.2	✔	6.7	✔	0.2	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
22. `resource-analysis`	✔	2.9	$\langle cs{:}0, ls\rangle$	**M**	65.6	✔	1.3	✔	2.6	✔	0.2	$\langle cs{:}0, tp{:}F, io{:}F, ls\rangle$
23. `sum-appendix`	✔	1.3	$\langle cs{:}0, ls\rangle$	☢	0.1	✔	1.2	✔	1.8	✔	0.0	$\langle cs{:}0, tp{:}F, io{:}T, ls\rangle$
geomean for ✔'s:		**1.9**			**1.6**		**6.7**		**5.1**		**0.9**	

In addition to the above, we also provide 23 unsafe *benchmarks.*
evDrift *analyzed all of them (using the Config in the last column above) in 273s.*

verify several benchmarks like `lics18-web`, `higher-order`, etc. that Drift cannot. (2) **ev**Drift establishes concrete relationships between program variables and accumulator variables leading to increased precision especially in resource-analysis-like benchmarks. (3) **ev**Drift's abstract domain adds some inbuilt disjunctivity reasoning that learns different relationships for different final states. This adds efficiency and precision to **ev**Drift's analysis as it is able to verify some benchmarks, like `disj-gte` that has if-then-else statements, without using trace partitioning. Furthermore, this disjunctivity enables evDrift to effectively track how individual states transition in every function, which is especially useful for higher-order programs like `intro-ord3` and `market`.

Due to their similarities, **ev**Drift also inherits a few limitations from Drift. **ev**Drift fails to verify `lics18-hoshrink`, which involves non-linear invariants presently not expressible by any of the abstract domains in the Apron library. Drift evaluates all nodes repeatedly until convergence of the whole program. We think it is possible to avoid such repetitions by selecting the order in which nodes are evaluated based on the data-flow dependency graph. We expect this to bring significant improvements in practice.

We also evaluated the impact of trace partitioning on the precision and performance of both Drift and **ev**Drift. For lack of space, the details can be found in the extended version [46] of this paper. In summary, **ev**Drift is able to verify two more benchmarks with trace partitioning, yet it slows down by 0.8×. Finally, since **ev**Drift is parametric on abstract domains, it can apply specialized domains when program features necessitate it. For example, the first program in Fig. 6 involves the `mod` operator. **ev**Drift is able to verify this example and four others examples (see benchmarks `last-ev-even`, `order-irrel`, `sum-of-ev-even` and `temperature` included with the release of **ev**Drift).

9 Conclusion

We have introduced the first abstract interpretation for inferring types and effects of higher-order programs. The effect abstract domain disjunctively organizes summaries (abstractions) of partitions of possible event trace prefixes around the concrete automaton states they reach. Our effects are captured in a refinement type-and-effect system and we described how to automate their inference through abstract interpretation. We then showed that our implementation **ev**Drift enables numerous new benchmarks to be verified (or enables faster verification by 6.3× on Drift-verifiable programs), as compared with prior effect-less tools (Drift, RCaml/Spacer, MoCHi, and ReTHFL) which require translations to encode effects.

Related work. We discussed some related works in Sec. 1 and as relevant throughout the paper. We now remark in some more detail and mention further related works. The work of Pavlinovic et al. [49] is the most related, but their type system does not include effects or automata, nor do they support any of our new benchmarks. However, we have been inspired by their work and build on aspects of their type system, abstract interpretation and implementation.

Several prior works have explored systems for reasoning about sequential effects such as thread synchronization [12], heap mutation [60], producer effects [61], and temporal properties [21, 36, 44] as well as unified frameworks for such systems [15, 16, 26]. Notably, our accumulative type and effect system bears similarity with instances of Gordon's polymorphic type and effect system [15, 16]. Similar to our work, his framework is parametric in an algebraic structure of effects, a so-called *effect quantale*, that abstracts from how effects are accumulated along and across program traces. However, the focus of [15, 16] is on developing the meta-theory of such type systems rather than the problem of practical type and effect inference. So there are some key technical differences that stem from our focus on the latter problem. Notably, sequential composition of effects in an effect quantale must distribute over joins in both arguments. In contrast, the effect extension operator in our work must only be a monotone upper-approximation of trace extension (Eq. (1)), a weaker requirement that gives more flexibility when designing abstract domains for effect inference. We exploit this flexibility in our implementation to trade precision for efficiency. Gordon and Yun [17] explore constraint-based type inference and error localization algorithms for effect quantales, but they do not consider any instances of the general type system that feature dependent effects and type refinements comparable to ours.

Zhou et al. [68] propose Hoare Automata Types (HAT), which augment a refinement type system with an automata-based representation of pre- and post conditions for tracking sequential effects. Unlike our work, which focuses on effect inference, their work provides an algorithm for checking user-provided type and effect annotations. The user expresses temporal effects in linear temporal logic (LTL) on finite traces (respectively, symbolic regular languages) [14]. These formulas are then compiled to a variant of symbolic automata [9], enabling SMT-based type checking. The compilation to automata requires that the underlying symbolic automata class is closed under effective complementation. This limits expressivity. Notably, it rules out accumulator registers like those supported by SAA. In contrast, to enable automatic effect inference, our approach approximates the rich SAA semantics by abstract interpretation using a broad class of abstract domains (such as polyhedra, which are not closed under complementation). That said, HAT provide support for rich properties of algebraic data types, which our current implementation does not yet handle. Though, we see the extension of the analysis with support for algebraic data types as mostly orthogonal to the handling of effects.

Nguyen et al. [45] present a method for verifying contracts of stateful untyped higher-order programs. The analysis uses symbolic execution and relies on a form of predicate abstraction to obtain refinement predicates for over-approximating the program semantics. Unlike our whole program analysis, this approach is compositional, enabling the verification of program components. However, the analysis requires user-provided contracts to enable this compositional analysis.

Hofmann and Chen [21] discuss abstractions of Büchi automata, building their abstractions by using equivalence classes and subsequences of traces to separately summarize the finite and the infinite traces. They then discuss a Büchi type & effect system, but it is not accumulative in nature, nor do they provide an implementation. Murase et al. [43] described a method of verifying temporal properties of higher-order programs through the Vardi [64] reduction to fair termination. We considered using some of their benchmarks, however none were suitable because the overlap between their work and ours is limited for two reasons: (i) they focus on verifying both liveness and safety while we only verify safety properties and (ii) we support expressive SAA-based safety properties, which they do not support. RCaml is a verifier for OCaml-like programs with refinement types, is based on extensions of Constrained Horn Clauses and is part of CoAR. RCaml was developed as part of several works [30, 37, 54]. Kobayashi's [32, 33] higher-order model checking, notably the approaches based on counterexample-guided abstraction refinement, CEGAR, is orthogonal to our proposed analysis. We make different trade-offs both in terms of algorithmic techniques as well as theoretical guarantees. In particular, unlike CEGAR-based approaches, our analysis is guaranteed to always terminate (although, CEGAR-based approaches can also show that the program is unsafe). Recently, Yamada et al. [67] explored the relationship between Dijkstra Monads [60] and the higher-order fixpoint logic HFL(Z) [34] for automated verification of higher-order programs [35]. Their proposed verification approach has been implemented in the tool ReTHFL [27], which we have included in our experimental comparison.

We have focused on events/effects that simply emit a value (**ev** v) that is unobservable to the program, and merely appears in the resulting event trace. By contrast, numerous recent works are focused on higher-order programming languages with *algebraic effects and their handlers*. Such features allow programmers to define effects in the language, and create exception-like control structures for how to handle the effects. Dal Lago and Ghyselen [8] detail semantics and model checking problems for higher-order programs that have effects such as references, effect handlers, etc. Although this work is quite general, it focuses on semantics and decidability, does not specifically target symbolic accumulator properties, and does not include an implementation. Kawamata et al. [30] discuss a refinement type system for algebraic effects and handlers that supports changes to the so-called "answer type."

Future work. A natural next direction is to automate verification of properties extend beyond safety to liveness specified by, say, Büchi automata or other infinite word automata, perhaps with an accumulator. Such an extension would require infinite trace semantics for the programming language and type & effect system (e.g. [36]), as well as a combination of both least and greatest fixpoint reasoning for abstract interpretations.

Data-Availability Statement

The software developed for this work and the full set of benchmarks used in our evaluation are available at Zenodo [47]

Acknowledgments

We thank the OOPSLA 2025 anonymous reviewers for their constructive comments on this draft. We are also grateful to Hiroshi Unno for generously providing his guidance on the use of RCaml which we employ in our evaluation. The work is supported by National Science Foundation under the grant agreements 2315363, 2107169, 2008633, and 2304758.

References

[1] Roberto Bagnara, Katy Louise Dobson, Patricia M. Hill, Matthew Mundell, and Enea Zaffanella. 2006. Grids: A Domain for Analyzing the Distribution of Numerical Values. In *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4407)*, Germán Puebla (Ed.). Springer, 219–235. doi:10.1007/978-3-540-71410-1_16

[2] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* 72, 1-2 (2008), 3–21. doi:10.1016/J.SCICO.2007.08.001

[3] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. 2004. Deriving Filtering Algorithms from Constraint Checkers. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3258)*, Mark Wallace (Ed.). Springer, 107–122. doi:10.1007/978-3-540-30201-8_11

[4] D. Beyer. 2024. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *Proc. TACAS (3) (LNCS 14572)*. Springer, 299–329. doi:10.1007/978-3-031-57256-2_15

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2007. A Static Analyzer for Large Safety-Critical Software. *CoRR* abs/cs/0701193 (2007). arXiv:cs/0701193 http://arxiv.org/abs/cs/0701193

[6] Michael Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. *Proc. ACM Program. Lang.* 8, POPL (2024), 2099–2128. doi:10.1145/3632912

[7] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96. doi:10.1145/512760.512770

[8] Ugo Dal Lago and Alexis Ghyselen. 2024. On Model-Checking Higher-Order Effectful Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 2610–2638. doi:10.1145/3632929

[9] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. 2019. Symbolic Register Automata. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 3–21. doi:10.1007/978-3-030-25540-4_1

[10] Katy Louise Dobson. 2008. *Grid domains for analysing software.* Ph. D. Dissertation. University of Leeds, UK. http://etheses.whiterose.ac.uk/1370/

[11] Kostas Ferles, Jon Stephens, and Isil Dillig. 2021. Verifying correct usage of context-free API protocols. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. doi:10.1145/3434298

[12] Cormac Flanagan and Shaz Qadeer. 2003. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, Ron Cytron and Rajiv Gupta (Eds.). ACM, 338–349. doi:10.1145/781131.781169

[13] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. doi:10.1145/113445.113468

[14] Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, Francesca Rossi (Ed.). IJCAI/AAAI, 854–860. http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997

[15] Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:31. doi:10.4230/LIPICS.ECOOP.2017.13

[16] Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 4:1–4:79. doi:10.1145/3450272

[17] Colin S. Gordon and Chaewon Yun. 2023. Error Localization for Sequential Effect Systems. In *Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14284)*, Manuel V. Hermenegildo and José F. Morales (Eds.). Springer, 343–370. doi:10.1007/978-3-031-44245-2_16

[18] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013. Software Model Checking for People Who Love Automata. 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, 36–52. doi:10.1007/978-3-642-39799-8_2

[19] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 357–370. doi:10.1145/1926385.1926427

[20] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 359–373. doi:10.1145/3009837.3009842

[21] Martin Hofmann and Wei Chen. 2014. Abstract interpretation from Büchi automata. In *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), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 51:1–51:10. doi:10.1145/2603088.2603127

[22] Atsushi Igarashi and Naoki Kobayashi. 2005. Resource usage analysis. *ACM Trans. Program. Lang. Syst.* 27, 2 (2005), 264–313. doi:10.1145/1057387.1057390

[23] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 661–667. doi:10.1007/978-3-642-02658-4_52

[24] Ranjit Jhala. 2025. LiquidHaskell. https://ucsd-progsys.github.io/liquidhaskell/

[25] Michael Kaminski and Nissim Francez. 1994. Finite-Memory Automata. *Theor. Comput. Sci.* 134, 2 (1994), 329–363. doi:10.1016/0304-3975(94)90242-9

[26] Shin-ya Katsumata. 2014. Parametric effect monads and semantics of effect systems. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 633–646. doi:10.1145/2535838.2535846

[27] Hiroyuki Katsura, Naoki Iwayama, Naoki Kobayashi, and Takeshi Tsukada. 2020. A New Refinement Type System for Automated $\nu\mathrm{HFL}_{\mathbb{Z}}$ Validity Checking. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 86–104. doi:10.1007/978-3-030-64437-6_5

[28] Hiroyuki Katsura, Naoki Iwayama, Naoki Kobayashi, and Takeshi Tsukada. 2025. ReTHFL. https://github.com/hopv/rethfl

[29] Hiroyuki Katsura, Naoki Iwayama, Naoki Kobayashi, and Takeshi Tsukada. 2025. Web interface for ReTHFL. https://www-kb.is.s.u-tokyo.ac.jp/~koba/nuhfl/

[30] 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. doi:10.1145/3633280

[31] Kenneth L. Knowles and Cormac Flanagan. 2009. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, Thorsten Altenkirch and Todd D. Millstein (Eds.). ACM, 27–38. doi:10.1145/1481848.1481853

[32] Naoki Kobayashi. 2009. Types and higher-order recursion schemes for verification of higher-order programs. 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, 416–428. doi:10.1145/1480881.1480933

[33] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 222–233. doi:10.1145/1993498.1993525

[34] Naoki Kobayashi, Kento Tanahashi, Ryosuke Sato, and Takeshi Tsukada. 2023. HFL(Z) Validity Checking for Automated Program Verification. *Proc. ACM Program. Lang.* 7, POPL (2023), 154–184. doi:10.1145/3571199

[35] Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. 2018. Higher-Order Program Verification via HFL Model Checking. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 711–738. doi:10.1007/978-3-319-89884-1_25

[36] Eric Koskinen and Tachio Terauchi. 2014. Local temporal reasoning. In *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), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 59:1–59:10. doi:10.1145/2603088.2603138

[37] Satoshi Kura and Hiroshi Unno. 2024. Automated Verification of Higher-Order Probabilistic Programs via a Dependent Refinement Type System. arXiv:2407.02975 [cs.LO] https://arxiv.org/abs/2407.02975

[38] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 392–411. doi:10.1007/978-3-642-54833-8_21

[39] Ugo Dal Lago and Alexis Ghyselen. 2024. On Model-Checking Higher-Order Effectful Programs. *Proc. ACM Program. Lang.* 8, POPL (2024), 2610–2638. doi:10.1145/3632929

[40] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 47–57. doi:10.1145/73560.73564

[41] Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 5–20. doi:10.1007/978-3-540-31987-0_2

[42] Antoine Miné. 2007. The Octagon Abstract Domain. *CoRR* abs/cs/0703084 (2007). arXiv:cs/0703084 http://arxiv.org/abs/cs/0703084

[43] Akihiro Murase, Tachio Terauchi, Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2016. Temporal verification of higher-order functional programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 57–68. doi:10.1145/2837614.2837667

[44] Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 759–768. doi:10.1145/3209108.3209204

[45] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2018. Soft contract verification for higher-order stateful programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 51:1–51:30. doi:10.1145/3158139

[46] Mihai Nicola, Chaitanya Agarwal, Eric Koskinen, and Thomas Wies. 2025. Abstract Interpretation of Temporal Safety Effects of Higher Order Programs [Extended Version]. (2025). arXiv:2408.02791 [cs.PL]

[47] Mihai Nicola, Chaitanya Agarwal, Eric Koskinen, and Thomas Wies. 2025. The evDrift Verifier. Zenodo. doi:10.5281/zenodo.16602546

[48] Lasse R. Nielsen. 2001. A Selective CPS Transformation. In *Seventeenth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2001, Aarhus, Denmark, May 23-26, 2001 (Electronic Notes in Theoretical Computer Science, Vol. 45)*, Stephen D. Brookes and Michael W. Mislove (Eds.). Elsevier, 311–331. doi:10.1016/S1571-0661(04)80969-1

[49] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2021. Data flow refinement type inference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–31. doi:10.1145/3434300

[50] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 46–57. doi:10.1109/SFCS.1977.32

[51] Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.* 29, 5 (2007), 26. doi:10.1145/1275497.1275501

[52] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. doi:10.1145/1375581.1375602

[53] Ryosuke Sato. 2025. MoCHi. https://github.com/hopv/mochi

[54] 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. doi:10.1145/3571264

[55] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 46–59. doi:10.1145/3009837.3009885

[56] Christian Skalka and Scott F. Smith. 2004. History Effects and Verification. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings (Lecture Notes in Computer Science, Vol. 3302)*, Wei-Ngan Chin (Ed.). Springer, 107–128. doi:10.1007/978-3-540-30477-7_8

[57] Christian Skalka, Scott F. Smith, and David Van Horn. 2008. Types and trace effects of higher order programs. *J. Funct. Program.* 18, 2 (2008), 179–249. doi:10.1017/S0956796807006466

[58] sosy lab. 2025. benchexec. https://github.com/sosy-lab/benchexec

[59] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 555–571. doi:10.1109/SP40001.2021.00085

[60] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 387–398. doi:10.1145/2491956.2491978

[61] Ross Tate. 2013. The sequential semantics of producer effect systems. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 15–26. doi:10.1145/2429069.2429074

[62] Tachio Terauchi. 2010. Dependent types from counterexamples. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 119–130. doi:10.1145/1706299.1706315

[63] Hiroshi Unno. 2025. CoAR. https://github.com/hiroshi-unno/coar/

[64] Moshe Y. Vardi. 1987. Verification of Concurrent Programs: The Automata-Theoretic Framework. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society, 167–176.

[65] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, 39–51. doi:10.1145/2633357.2633366

[66] Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. doi:10.1145/292540.292560

[67] Risa Yamada, Naoki Kobayashi, Ken Sakayori, and Ryosuke Sato. 2025. On the Relationship between Dijkstra Monads and Higher-Order Fixpoint Logic. In *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 15695)*, Viktor Vafeiadis (Ed.). Springer, 402–428. doi:10.1007/978-3-031-91121-7_16

[68] Zhe Zhou, Qianchuan Ye, Benjamin Delaware, and Suresh Jagannathan. 2024. A HAT Trick: Automatically Verifying Representation Invariants using Symbolic Finite Automata. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1387–1411. doi:10.1145/3656433

Received 2025-03-25; accepted 2025-08-12