Inferring Accumulative Effects of Higher Order Programs

ANONYMOUS AUTHOR(S)

Many temporal safety properties of higher-order programs go beyond simple event sequencing and require an automaton register (or "accumulator") to express, such as input-dependency, event summation, resource usage, ensuring equal event magnitude, computation cost, etc. Some steps have been made towards verifying more basic temporal event sequences via reductions to fair termination or some input-dependent properties through deductive proof systems. However, there are currently no automated techniques to verify the more general class of register-automaton safety properties of higher-order programs.

We introduce an abstract interpretation-based analysis to compute dependent, register-automata effects of recursive, higher-order programs. We capture properties of a program's effects in terms of automata that summarizes the history of observed effects using an accumulator register. The key novelty is a new abstract domain for context-dependent effects, capable of abstracting relations between the program environment, the automaton control state, and the accumulator value. The upshot is a dataflow type and effect system that computes context-sensitive effect summaries. We demonstrate our work via a prototype implementation 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 effect-free programs, and demonstrate that our approach outperforms prior tools Drift, RCaml/PCSat, and MoCHi. Overall, across a set of 25 new benchmarks, Drift verified 11 benchmarks, RCaml/PCSat verified 3, MoCHi verified 8, and **ev**Drift verified 23; **ev**Drift also achieved a 25×, 3.4×, and 16× speedup over Drift, RCaml/PCSat, and MoCHi, respectively, on those benchmarks that both tools could solve.

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 [10, 33, 36, 43, 47], capable of validating merely (non-temporal) safety assertions. Subsequently, works focused on verifying termination of higher-order programs, e.g., [26].

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* (e.g. [28]), typically emitted as a side effect of a 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 [34], or as an automaton. Liveness properties apply to programs that may diverge, inducing infinite event traces. A first approach at temporal verification was through the celebrated reduction to fair termination [45]. Murase et al. [31] 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 [39], Skalka et al. [40] 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 [24] and Hofmann and Chen [15] showed that the effects component in a type-and-effect system $\Gamma \vdash e : \tau \& \varphi$ could consist of a temporal property φ to hold of the events generated by the reduction of expression e. This was combined with a dependent refinement system by Koskinen and Terauchi [24] and used with an abstraction of Büchi automata by Hofmann and Chen [15]. Nanjo et al. [32] 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 [27, 37] (see Sec. 9).

There are, however, 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, or that the last emitted integer is the largest one. Properties could depend on inputs (like some examples in Nanjo et al. [32]), or involve context-free-like properties such as the sum of production being equal to the sum of consumption. Despite the above discussed prior works, there are currently no automatic tools for verifying such kinds of temporal safety properties of higher-order programs with effects.

1.1 This Paper

We introduce a new route to automate temporal effect inference and verification of recursive higher-order programs through abstract interpretation over a novel *effect abstract domain*. To support the wider class of temporal safety properties mentioned, we first augment the property language to those expressible via a symbolic accumulator automaton (SAA). Our automaton model is inspired by the various notions of (symbolic) register or memory automata considered [3, 8, 19] and consists of a register "accumulator" (e.g., an integer or tuple of integers) that can remember earlier events, calculate summaries, etc. Symbolic accumulator automata capture properties such as maximum/minimum, summation, resource bounds, context-free protocols of stateful APIs, etc., in addition to simple event ordering properties expressible with a regular finite state automaton.

As a first step, we observe that verifying such properties of higher-order programs with effects can be reduced to verifying assertions of effect-less higher-order programs. We show that this can be done via one of at least two program transformations: (i) translating the program so that every expression reduces to a (value, effect prefix) tuple, where the second component accumulates the effects as a sort of ghost return value, or (ii) translating the program into continuation-passing style, accumulating the effect prefix as an argument. We later experimentally show that, although this theoretically enables higher-order safety verifiers (e.g. Drift, RCaml/PCSat, and MoCHi) to be applied to the effect setting, those tools do not exploit much of the property structure and ultimately struggle on the inherent product construction 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 $Q \rightarrow \wp(\mathcal{V} \times Env)$. captures the possible pairs of accumulator value and execution environment 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: (i) a summary of the program environment, e.g. constraints like $\mathtt{x} > \mathtt{y}$, (ii) a summary of the automaton accumulator, e.g. constraints like $\mathtt{acc} > 0$, and even (iii) relations between the two, e.g. $\mathtt{acc} > \mathtt{x} - \mathtt{y}$. 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. The effect abstract domain can naturally be instantiated using any of a variety of standard numerical domains such as polyhedra [2, 7, 38], octagons [30], etc.

We remark that, in addition to addressing a class of safety properties that previous tools do not support, to our knowledge, our work is the first use of abstract interpretation toward inferring temporal effects of higher-order programs. In light of the promising results presented here, in future work we aim to further explore how abstract interpretation can provide an alternative route toward automating other classes of properties of higher-order programs such as liveness.

1.2 Challenges & Contributions

To pursue the effect abstract domain, we address a 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. Unfortunately existing type and effect systems [15, 24, 32] are not suitable because their judgments of the form $\Gamma \vdash e : \tau \& \varphi$ do not involve event trace prefixes in their context: instead φ describes the effects of e alone, without information about what effects preceded the evaluation of e. While this makes the type system more compositional, it also makes effect inference more difficult because one has to analyze e for any possible prefix trace. In terms of our automata-based effect domain, it corresponds to analyzing e for an arbitrary initial control state and accumulator value. We thus present a new 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.

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 [33]. We present an extension to *effectful* programs through a translation-oriented embedding of programs with effects to effect-free programs. 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 effort to find the closest basis for comparison, we also implemented two translations (one via encoding effects in tuples; another via encoding effects as a CPS parameter) that reduce SAA verification of effect programs to assertion checking of effect-free programs (to which Drift, RCaml/PCSat, MoCHi, etc. can be applied). To improve the precision of our abstract interpretation, we also adapted the classical notion of *trace partitioning* [29] 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

25 new examples and adapting examples from the literature including summation/max-min/ examples [3, 8, 19], monotonicity examples, programs with temporal event sequences [24, 32], resource analysis [13, 14, 17], and an auction smart contract [42].

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/PCSat, and MoCHi 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 23 out of the 25 benchmarks, which is 12, 20, and 15 more than Drift, RCaml/PCSat, and MoCHi, respectively, (with our tuple translation) could verify. Furthermore, **ev**Drift achieved a speedup of 25×, 3.4×, and 16× over Drift, RCaml/PCSat, and MoCHi, 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.*

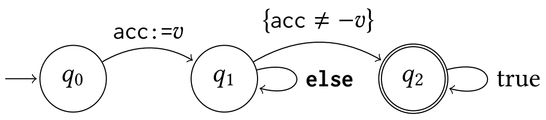
2 OVERVIEW

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 [33], 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 $\cup_{x\in\mathbb{Z}}\{\{x,-x\}\}$, 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. There is an initial control state q_0, from which point, whenever an event 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: $\cup_{x\in\mathbb{Z}}\{x(-x)^*\}$, which permits arbitrarily many $-x$ events after x (including none).

Naïve 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

```
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)
```

to an automaton [11]) 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 function to the right, which consumes the current automaton configuration, and a next event value v and returns the next configuration.

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 =
  let (q',acc') =
    if (n <= 0) then ev_step q acc (-t)
    else busy_prod q acc (n - 1) t
  in (q',acc')
```

```
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.

The above example is recursive and we are also interested in programs that are higher-order (though the above simple example is only first-order), which limits the field of applicable existing tools. One example is the Drift tool which uses a dependent type system and abstract interpretation to verify safety properties of higher-order recursive programs [33]. We implemented the above translation (details in Appendix C.3). As part of its approach, Drift then converts the program to lambda calculus expressions, leading to a program that is much larger (roughly 75 lines of code). For this example Drift is able to verify the product encoding in 11.5s. RCaml/PCSat (part of CoAR[12]), another fairly mature tool that can also verify assertions of higher order programs [20, 25, 37], took 3.2s to verify it. Meanwhile, MoCHi [16], a software model checker based on higher-order recusion schemes [22, 23], needed 49.2s for verifying the same tuple translation of the program. By contrast, this paper will introduce an abstraction that can verify this program/property in 0.4s.

The problem. Although this example tuple product reduction can be verified by existing tools, unsurprisingly, the approach does not scale well with neither one of the considered tools. Let us examine another example called temperature, shown in the top left of Fig. 1, that is only slightly more involved yet causes Drift to timeout after 900s when the tuple product reduction is used. We will describe a technique and tool that can instead verify this example in 35s.

The temperature example in Fig. 1 can be thought of as a simple model of a thermostat, which can either be in a heating mode (when input v is even) or a cooling mode (when input v is odd). Function f is initially called with x (i.e. v), and a positive integer pos, and negative integer neg. Whenever x is even, an event of value pos is emitted and the temperature is increased. Whenever x is odd, the value neg is emitted and neg is decreased. At each recursive call of f, x decreases by 2, so it will always either be even or odd. Thus, the event traces of the program are: $\cup_{x\in\mathbb{Z}^+}\{x, x+1, x+2, \ldots\} \bigcup \cup_{x\in\mathbb{Z}^-}\{x, x-1, x-2, \ldots\}$, i.e. any sequence of a monotonically increasing positive integer or any sequence of a monotonically decreasing negative integer. This property is captured by the accumulator automaton illustrated in the top right of Fig. 1. If the first event is positive, the upper arc is taken, and the value of the event is remembered in the accumulator. Thereafter, as long as each successive event is greater than the previous, the automaton loops at q_1. Dually, if the initial event is negative and subsequent events monotonically decrease, the automaton loops at q_2. If ever the monotonic increase/decrease is violated, a transition is taken to q_{err}.

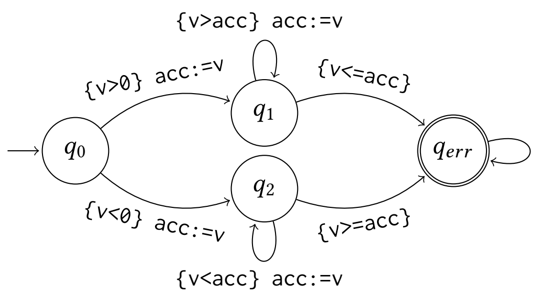
The struggle. A naïve translation-based reduction to existing safety verification tools for higher-order programs does not fare well and the reason is twofold. First, there is a blowup in the size of

Input Program:

```
let rec f x pos neg =
  if x % 2 = 0 then (ev pos)ⓟ
  else              (ev neg)ⓝ;
  if (x <= 0) then 0
  else f (x-2) (pos+1) (neg-1)

let temperature (v p n:int) =
  if p > 0 && n < 0 then
    f v p n
  else 0
```

Input Property: Initially, acc= 0.



Computed Effect Abstractions :

Location ⓟ :

$q_0^{\textcircled{p}} \mapsto \bot$

$q_1^{\textcircled{p}} \mapsto \mathsf{x\%2} = 0 \land \mathsf{pos} = \mathsf{acc} > 0$

$q_2^{\textcircled{p}} \mapsto \bot$

$q_{err}^{\textcircled{p}} \mapsto \bot$

Location ⓝ :

$q_0^{\textcircled{n}} \mapsto \bot$

$q_1^{\textcircled{n}} \mapsto \bot$

$q_2^{\textcircled{n}} \mapsto \mathsf{x\%2} = 1 \land \mathsf{neg} = \mathsf{acc} < 0$

$q_{err}^{\textcircled{n}} \mapsto \bot$

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

Fig. 1. In the top left is the input program temperature. The top right illustrates the symbolic accumulator automaton property specifying that, roughly, the system is either in heating mode (q=1) or cooling mode (q=2), with the events monotonically increasing in heating mode and decreasing in cooling mode. At the bottom is the effect abstraction that we automatically compute in this paper, discussed in this section.

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 naïve 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 $\mathbf{ev}\ v$ is reduced: $\wp(\mathbb{Z}^* \times Env) \xrightarrow{\mathbf{ev}\ v} \wp(\mathbb{Z}^* \times Env)$. For the above example, a concrete sequence of states and transitions could be the following:

$$(\epsilon, [main, \mathsf{v}:42, \mathsf{p}:3,\ \mathsf{n}:-8]) \rightsquigarrow (\epsilon, [f, \mathsf{x}:42, \mathsf{pos}:3,\ \mathsf{neg}:-8]) \xrightarrow{\mathbf{ev}\ 3} (3, [f, \mathsf{x}:42, \mathsf{pos}:3,\ \mathsf{neg}:-8])$$
$$\rightsquigarrow (3, [f, \mathsf{x}:40, \mathsf{pos}:3,\ \mathsf{neg}:-8]) \xrightarrow{\mathbf{ev}\ 3} (3\cdot 3, [f, \mathsf{x}:40, \mathsf{pos}:3,\ \mathsf{neg}:-8]) \cdots$$

(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 starting environment, the trace will accumulate the event sequence is $3 \cdot 3 \cdots$.

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. 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 categorize 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 $\mathbf{ev}\ v$ 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 `temperature` example, when an execution involves positive temperatures and there is an event trace prefix such as $1 \cdot 2 \cdot 3$, then the following lists some of the effects at body of `f` per each q:

$$q_1 \mapsto \{(1, (\mathsf{x}:42, \mathsf{pos}:1, \mathsf{neg}:\_)), (2, (\mathsf{x}:40, \mathsf{pos}:2, \mathsf{neg}:\_)), \ldots\}, \quad q_0 \mapsto \emptyset, \quad q_2 \mapsto \emptyset, \quad q_{err} \mapsto \emptyset.$$

Above q_0 is not reachable because at the point when the program reaches location ⓟ, at least one event must already have been emitted. q_1 is only reachable with event sequences that have at least one positive event and in the corresponding environments, `x` will be even, and `pos` will be equal to the most recent event value. Finally, there are also no event sequences that reach q_2 or q_{err}.

Abstract relations with the accumulator. Thus far we associate event sequence and environment pairs per control state, however, 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** `pos` location denoted ⓟ, our summary for $q_1^{ⓟ}$ reflects the positive, monotonically increasing sequences and our summary for $q_2^{ⓝ}$ reflects the negative, monotonically decreasing sequences. The automaton specifies if ever this is violated it will transition to q_{err}. The program is safe because at every location ⓘ, we compute $q_{err}^{ⓘ} \mapsto \bot$.

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 $(\mathbf{ev}\ \mathsf{pos})^{\textcircled{p}}$ expression to ensure events are positive and monotonically increasingly. First, let $\phi^{mono}_{acc}(\mathsf{pos})$ be shorthand for $\mathsf{pos} = \mathsf{acc} \wedge \mathsf{acc} > 0$, i.e., that the accumulator is equal to the program value of pos and that both are positive. Further, due to the recursive call, when we reach $(\mathbf{ev}\ \mathsf{pos})^{\textcircled{p}}$, we have that pos increases by one: $\mathsf{pos} \rightsquigarrow \mathsf{pos} + 1$. We thus obtain the following jugdment for $(\mathbf{ev}\ \mathsf{pos})^{\textcircled{p}}$:

$$\begin{array}{l}\Gamma; [q_1 \mapsto \phi^{mono}_{acc}(\mathsf{pos\text{-}1}), q_{err} \mapsto \bot, \ldots] \\ \vdash \mathbf{ev}\ \mathsf{pos} : ()\& \left[\begin{array}{l} q_1 \mapsto \boxed{[\phi^{mono}_{acc}(\mathsf{pos\text{-}1})] \odot [g : \mathsf{pos} > \mathsf{acc}]; [u : \mathsf{acc} := \mathsf{pos}]}, \\ q_{err} \mapsto \bot, q_0 \mapsto \bot, q_2 \mapsto \bot \end{array}\right]\end{array}$$

We focus on the result of computing the $\odot$ on the state q_1 to q_1 of the extended effect in the context of the judgment indicated by the boxed area. Note that the following sequence of entailments is valid: *guard* $\phi^{mono}_{acc}(\mathsf{pos\ -\ 1}) \implies pos - 1 = acc \implies g : \mathsf{pos} > \mathsf{acc}$. This means that the guard of the automaton transition from q_1 to q_1 must be satisfied by all the prefix traces in the context. The update of *acc* to *pos* reestablishes the $\phi^{mono}_{acc}(\mathsf{pos})$ at q_1. None of the guards of the other outgoing transitions from q_1 are satisfied, thus other states map again to $\bot$. In summary, 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 [33]) 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 `temperature` (and others among the 25 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 naïve product reductions to compare against tools for effect-free programs.

Evaluation (Sec. 8). **ev**DRIFT verifies `temperature` in 35s, whereas previous assertion-verifiers (combined with our translations) either timeout (DRIFT) or run out of memory. More generally, **ev**DRIFT verifies more examples and otherwise outperforms DRIFT, RCAML/PCSAT, and MoCHI by 25×, 3.4×, 16×, 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 $e \in \mathcal{E} ::= c \mid x \mid \lambda x.\, e \mid (e\ e) \mid \mathbf{ev}\ \ e$ and values $v \in \mathcal{V} ::= c \mid \lambda x.\, e$. Expressions e in the language consist of constant values $c \in Cons$ (e.g. integers and Booleans), variables $x \in Var$, function applications, lambda abstractions, and event expressions $\mathbf{ev}\ \ e_1$. We assume the existence of a dedicated unit value $\bullet \in Cons$. 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

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

Fig. 2. Reduction rules of operational semantics.

e with term e' while avoiding variable capturing. We further write $\mathsf{fv}(e)$ for the set of free variables occurring in e.

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

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: $E ::= [] \mid E\ e \mid v\ E \mid \mathbf{ev}\ E$. 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 $\mathbf{ev}\ v$ 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 [28] 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 [32, 39, 40]. 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 v'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 let $\mathsf{dom}(\beta)$ denote the set of variables $x \in \mathit{Var}$ that are constrained by β. Formally:

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

Examples of possible choices for $\mathcal{B}$ include standard relational abstract domains such as octagons and polyhedra. 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 ν refers to the value and x, y, z are the variables evaluated in the environments:

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

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 $\gamma^{\phi} \in \Phi \rightarrow \wp(\mathcal{V}^* \times Env)$. 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:

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

Intuitively, a function type $x : (\tau_2 \& \phi_2) \rightarrow \tau_1 \& \phi_1$ 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:

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

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

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

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:

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

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:

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

$$\textsc{s-base}\ \frac{\beta_1[\Gamma] \sqsubseteq^b \beta_2}{\Gamma \vdash \beta_1 <: \beta_2} \qquad \textsc{s-wit}\ \frac{\Gamma \vdash \tau' <: \tau \qquad \Gamma, y : \tau' \vdash \tau_1 <: \tau_2[y/x]}{\Gamma, y : \tau' \vdash \tau_1 <: \exists x : \tau.\, \tau_2} \qquad \textsc{s-exists}\ \frac{\Gamma, x : \tau \vdash \tau_1 <: \tau_2}{\Gamma \vdash \exists x : \tau.\, \tau_1 <: \tau_2}$$

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

Fig. 3. Semantic subtype relation.

Typing judgements. Our type system builds on existing refinement type systems with semantic subtyping [6, 21]. 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 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. 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:

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

We require that $\odot$ is monotone in both of its arguments. The following is an example judgment for the **ev** $(pos + 1)$ expression in the `temperature` example from Sec. 2:

$$\Gamma, pos : \tau; [..q_1 \mapsto acc = pos > 0] \vdash \textbf{ev}\ (pos + 1) : (unit \& [..q_1 \mapsto acc = pos + 1 > 0])$$

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

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

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

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

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

Fig. 4. Typing relation.

The effect to the left of the turnstile describes event prefixes associated with all executions leading to the evaluation of expression **ev** $(pos + 1)$. It states that q_1 is the only reachable state and the accumulator is equal to variable pos. Given that pos has the base type $\tau = \{\nu \mid \nu > 0\}$, we can derive that $(pos+1)$ has type $\tau' = \{\nu \mid \nu = pos+1\}$. The typing judgement states that for all executions the extended effect that accounts for the concatenated values represented by τ' preserves the invariant between the variable representing observable events at that location and the accumulator. When effects are drawn from the SAA-based abstract domain, the symbolic guard $[acc < pos + 1]$ must be satisfied before updating the accumulator to the new symbolic value $(pos + 1)$ of the event.

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 Apx. A, 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. $\phi \odot \tau \overset{\text{def}}{=} \{\langle \pi \cdot v, \rho \rangle \mid \langle v, \rho \rangle \in \tau \wedge \langle \pi, \rho \rangle \in \phi\}$. 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. 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 design 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, 8, 19]. 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 $\mathcal{U}$. 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 $[\![g]\!](v, a) = \mathtt{true}$ 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 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 and refer the reader to Appendix B for its full formalization. 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 represent 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 Galois connection between $\wp(\mathcal{V}^* \times Env)$ and Φ_C is materialized via its upper adjoint γ_C^ϕ, a concretization function mapping elements in Φ_C to event prefix-value pairs. Its element-wise definition ensures its monotonicity and the preservation of arbitrary meets.

A key operation on the effect domain is effect extension $\odot_C : \Phi_C \times \mathcal{B} \to \Phi_C$, which we define in the usual way as the most precise abstraction of its concrete counterpart. Its characterization relies on the totality of the transition relation of the automaton, and its soundness condition, imposed in §4, is immediate by construction and the properties of the Galois connections.

Abstract automata domain of dependent effects. To obtain a symbolic abstract domain that enables effective static analysis, we further abstract Φ_C. We proceed in two steps. The first abstraction partitions each element $\phi_c \in \Phi_C$ by the control location of the automaton state. That is we define $\Phi_R : (Q \to \wp(\mathcal{V} \times Env))$, ordered point-wise by inclusion. It is immediate to see that this step introduces no loss of precision. Furthermore, the Galois connection between Φ_C and Φ_R is, in fact, a lattice isomorphism. Let $\gamma_R^\phi : \Phi_R \to \Phi_C$ be the concretization function between the two domains, with its inverse $\alpha_R^\phi = \gamma_R^{\phi^{-1}}$ serving as the abstraction function. The corresponding effect extension operator $\odot_R : \Phi_R \times \mathcal{B} \to \Phi_R$ is then constructed without loss of precision from its counterpart on Φ_C using this Galois connection.

The second step abstracts the accumulator and environment components at each control location using a relational abstract domain. In fact, we can reuse the domain of base refinement types $\mathcal{B}$ from §4 for this purpose: 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 lift the concretization γ^b pointwise to a function $\Phi_{\mathcal{B}} \to \Phi_C$ by defining $\gamma_{\mathcal{B}}^\phi(\phi) = \gamma^b \circ \phi$.

Operations on Φ are defined in terms of operations on $\mathcal{B}$. We again focus on the extension operator, $\odot$. To ensure that $\odot$ maintains soundness, we define it in such a way that is satisfies the condition

$$\gamma_{\mathcal{B}}^\phi(\phi \odot \beta) \;\supseteq\; \gamma_{\mathcal{B}}^\phi(\phi) \odot_R \beta \;\; . \tag{1}$$

With this in place, soundness of $\odot$ is achieved by construction. Specifically, we formalize $\odot$ by expanding $\gamma_{\mathcal{B}}^\phi(\phi) \odot_R \beta$ to express it as a combination of possible transitions across states, considering guards and update functions on the automaton's symbolic transitions. Guards and updates are abstracted using an abstract interpreter that evaluates expressions over base refinement types. Further details, including the proof outline for the soundness argument are available in Appendix B.

6 AUTOMATED VERIFICATION

We now describe how to verify accumulation automaton properties of higher order programs, through automatic inference of types and effects. Our strategy builds on an existing data flow refinement type inference algorithm [33] based on abstract interpretation. At a high level, there are four steps: (i) translate the program so that prefix event traces π are symbolically represented at the syntactic level, (ii) extend refinement *types* and type inference to support *event sequences* using our novel effect abstract domain, (iii) the types of those event sequences then correspond to *effects* in our type and effect system in §4 and then (iv) ensure that at every program location ⓘ, the computed summary associates $\perp$ with every accepting state of the SAA that encodes the property of interest. We now describe these steps; for lack of space details are available in Appendix C.

(i) Translation. We translate the original effectful program into a language where **ev** e expressions are absent. Within this language, the prefix event traces π are encoded as event sequence values and are symbolically represented at the syntactic level. This function preserves the operational semantics while ensuring that the events emitted during the program's execution are carried through the computation. The procedure is somewhat straightforward, translating every expression e so that it produces a pair consisting of the value and event sequence obtained from evaluating e, effectively monadifying the effectful computation.

(ii) Inferring refinement types for event sequences. Data flow type inference, as described by [33], 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 Drift. The typing produced by the algorithm yields a valid typing derivation in a dependent refinement type system, which we refer to as the Drift type system.

In our instantiation of the Drift type system, we treat event sequences as values that can be manipulated directly by the program. The details of the inference algorithm itself are of no import for the present paper. What is relevant is that 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.).

(iii) Type and Effect. The inferred *types of encoded event sequences* correspond to *effects* of the events in the original program. 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. To bridge this gap, we relate the two type systems at the abstract level by showing that, from the typing derivation for a translated program produced by the soundness proof of [33], one can reconstruct a typing derivation in the types and effects system for the original effectful program. The key soundness Theorem C.1 and its proof are in Appendix C.4. The overall soundness then follows from Theorem 4.1.

Rather than describing the inference algorithm in detail, we can therefore focus on the instantiation of the Drift type system with the relevant primitive operations needed in our translation. Apart from not directly supporting effects, the Drift type system slightly deviates from ours in handling subtyping because it is designed to characterize the fixpoints of the abstract interpretation.

(iv) Verification. As discussed in Sec. 2, our abstract effect domain seeks to improve over a naïve approach of translating (via tuples or CPS) an input program/property of effects into an effect-free product program that carries its effect trace and employs existing assertion checking techniques [20, 23, 33]. 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.

Contrast to naïve product translation. The soundness of our type and effect inference follows a translation argument. Thus a keen reader may notice some conceptual similarities between the naïve product translation and the translation discussed in this section. We emphasize that these are absolutely not the same thing for a few reasons. First, we can take advantage of the fact that the abstract interpreter can be specialized to only analyze translated programs. That is, our abstract interpretation method here performs the translation *implicity* during its analysis of the (untranslated) program. As such it can fuse together what would otherwise be costly joins and projections needed for analysis of the naïve product. Second, the automaton transition function is not embedded directly in the program. Instead, the sequence concatenation operator is interpreted by the abstract effect domain. Finally, the abstract domain is organized around the concrete automaton states, providing direct disjunctivity in the abstract domain rather than requiring the abstract interpreter to have some other means of disjunctive reasoning.

7 IMPLEMENTATION, TRACE PARTITIONING, 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 [33] type inference tool, which builds on top of the APRON library [18] 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 two improvements to the dataflow abstract interpretation. First, the grid-polyhedra abstract domain [9]—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 [35] 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 abstract effect domain. (Note that these two 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 [35]. Our extended type inference algorithm uses a combination of abstract stacks and traces to distinguish between the different evaluation paths created by if-then-else expressions and function call sites leading up to a program node. However, naturally, this comes at the cost of processing some nodes of the program multiple times; we evaluate the impact of

`let rec order d c =`	`let rec spend n =`	`let rec reent d =`	`let rec compute vv bound inc =`
`if d > 0 then`	`ev (−1);`	`if d > 0 then`	`if vv = bound then 0 else`
`if d mod 2 = 0`	`if n <= 0 then 0`	`ev 1; (* Acq *)`	`ev vv;`
`then ev c`	`else spend (n−1)`	`if nondet() then`	`compute (inc vv) bound inc`
`else ev (−c);`		`reent (d−1);`	`let min_max v =`
`order (d − 2) c`	`let _ (gas n:int) =`	`ev −1 (* Rel *)`	`let f = (fun t ->`
`else 0`	`if gas >= n`	`else skip`	`if v>=0 then t−1 else t+1) in`
`let _ (dd cc:int) =`	`then`	`let _ d =`	`if v>=0`
`order dd cc`	`(ev gas; spend n)`	`reent d;`	`then compute v (−1 * v) f`
	`else 0`	`ev −1 (* Rel *)`	`else compute v (−1 * v) f`
Only "c" or "-c" events	$\sum_i^N \leq$ gas	# Rel ≤ # Acq	$\forall i > 0. -v < \pi[i] < v$

Fig. 5. Further examples of our benchmarks. (See the supplement for sources and automata specifications.)

trace partitioning in Sec. 8. We also give more details about our trace-partitioning implementation in Apx. D.

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 25 benchmarks to SV-COMP [4]. Figure 5 lists some of them. These benchmarks test our tool to verify a variety of SAA properties like (left-to-right in Fig. 5) 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, 8, 19]. We also include an auction smart contract [42] and adapt some example programs proposed in [32]. These benchmarks involve verification of amortized analysis [13, 14, 17] 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

All our experiments were conducted on an x86_64 AMD EPYC 7452 32-Core Machine with 125 Gi memory. We used BenchExec [41] for our experiments to ensure precise measurement of time and memory allocation for each run. In our experiments, 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-partioning on efficiency and accuracy?

Comparing our approach with other methods. To our knowledge there are no tools that can verify SAA properties of higher-order programs with effects. So to perform some comparison, we combine our translation reduction with tools that can verify assertions of higher-order (effect-free) programs. We consider three somewhat mature tools: Drift (discussed in Sec. 6 and Apx. C.4), RCaml/PCSat[1] and MoCHi [16] (see Secs. 1 and 9 for discussions of other works and tools). RCaml/PCSat is based on extensions of Constrained Horn Clauses, is part of CoAR [12], and is built on top of several prior works [20, 25, 37]. MoCHi[23] is a CEGAR-style software model checker based on higher-order recursion schemes and relies on either interpolating theorem provers or an ICE-based

[1]We use the RCaml/PCSat CoAR config, i.e., `config/solver/rcaml_wopp_sapcer.json`.

solver of Constrained Horn Clauses for predicate discovery. We also considered LiquidHaskell [46], which includes an implementation [44]. 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.

In our **ev**Drift experiments, we run all our benchmarks using several configurations: various combinations of context sensitivity, trace partitioning, numerical abstract domains, etc. Context sensitivity 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 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 and **ev**Drift *with* trace partitioning. 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 [33] 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], 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. 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 Apx. F found in the supplement.

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 11 of the 25 benchmarks, while **ev**Drift could verify 23. Across all benchmarks that Drift could solve, it had a geomean of 7.5s. Across all benchmarks that **ev**Drift could solve, it had a geomean of 1.9s. For those benchmarks that *both* Drift and **ev**Drift could verify, **ev**Drift was 25× faster. RCaml/PCSat successfully verified 3 out of 25 benchmarks, with a geomean of 0.6s across these, and was unable to verify the others due to either imprecision, timeout, or memory blowup. Finally, MoCHi verified 8 benchmarks for which it reported a geomean of 11.2s. However, 5 of our benchmarks (indicated in Table 1) use the `mod` operation that MoCHi soundly approximates by treating it as an interpreted function, and that leads to imprecision. 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.

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.

	Drift (via tuple reduction)				RCaml			MoCHi			evDrift			
Bench	**Res**	**CPU**	**Mem**	**Config.**	**Res**	**CPU**	**Mem**	**Res**	**CPU**	**Mem**	**Res**	**CPU**	**Mem**	**Config.**
1. `all-ev-pos`	✔	2.6	22.3	$\langle cs{:}0, tp{:}F, ls\rangle$	✔	0.5	72.6	✔	1.3	119.5	✔	0.2	8.4	$\langle 0, F, ls\rangle$
2. `auction`	?	244.3	195.7	$\langle cs{:}1, tp{:}F, oct\rangle$	**M**	40.3	1000.0	**T**	635.6	750.6	✔	1.6	15.9	$\langle 1, F, ls\rangle$
3. `binomial_heap`	☢	0.4	11.5	$\langle cs{:}0, tp{:}F, pg\rangle$	☢	0.3	37.9	?	9.7	165.1	✔	6.4	40.7	$\langle 0, F, ls\rangle$
4. `concurrent_sum`	✔	6.2	18.4	$\langle cs{:}1, tp{:}F, ls\rangle$	**M**	12.1	1000.0	**T**	636.2	834.1	✔	0.5	12.2	$\langle 1, F, ls\rangle$
5. `depend`	✔	0.1	8.6	$\langle cs{:}1, tp{:}F, ls\rangle$	✔	0.1	45.9	✔	0.5	60.4	✔	0.0	5.2	$\langle 1, T, ls\rangle$
6. `disj`$^+$	?	338.2	279.2	$\langle cs{:}1, tp{:}F, st\rangle$	☢	0.1	22.9	?	38.7	335.9	✔	9.6	21.3	$\langle 0, F, pg\rangle$
7. `disj-gte`	?	698.0	264.2	$\langle cs{:}1, tp{:}F, oct\rangle$	☢	0.1	22.9	✔	374.1	1004.5	✔	6.4	28.1	$\langle 1, F, ls\rangle$
8. `disj-nondet`	?	555.4	160.1	$\langle cs{:}1, tp{:}F, ls\rangle$	☢	0.1	23.5	**T**	635.6	1223.1	✔	3.6	52.4	$\langle 0, F, ls\rangle$
9. `higher-order`	✔	28.9	34.4	$\langle cs{:}1, tp{:}F, ls\rangle$	**M**	18.9	1000.0	✔	17.6	224.4	✔	0.6	12.6	$\langle 0, F, ls\rangle$
10. `last-ev-even`$^+$	?	74.5	181.0	$\langle cs{:}1, tp{:}F, oct\rangle$	☢	0.0	19.0	?	0.9	132.5	✔	9.6	20.4	$\langle 1, T, pg\rangle$
11. `lics18-amortized`	☢	5.6	76.9	$\langle cs{:}1, tp{:}F, ls\rangle$	**T**	931.0	913.4	**T**	783.5	1577.1	✔	41.6	44.6	$\langle 1, F, ls\rangle$
12. `lics18-hoshrink`	**T**	900.3	127.1	$\langle cs{:}1, tp{:}F, pg\rangle$	☢	0.1	19.6	?	30.9	206.0	?	1.4	15.6	$\langle 1, T, oct\rangle$
13. `lics18-web`	**T**	931.0	207.9	$\langle cs{:}1, tp{:}F, oct\rangle$	☢	0.5	43.7	✔	18.9	481.1	✔	35.5	51.5	$\langle 0, F, ls\rangle$
14. `market`	**T**	931.0	149.9	$\langle cs{:}1, tp{:}F, oct\rangle$	☢	0.8	73.1	**T**	596.7	1284.6	?	59.6	65.4	$\langle 1, F, st\rangle$
15. `max-min`	**T**	900.2	107.8	$\langle cs{:}0, tp{:}F, pg\rangle$	☢	0.1	23.6	**T**	932.1	1064.1	✔	90.0	80.3	$\langle 1, T, ls\rangle$
16. `monotonic`	✔	16.3	30.6	$\langle cs{:}1, tp{:}F, ls\rangle$	**T**	930.9	464.8	✔	17.9	508.1	✔	1.1	14.2	$\langle 1, F, ls\rangle$
17. `nondet_max`	✔	61.2	95.1	$\langle cs{:}1, tp{:}F, ls\rangle$	☢	0.0	19.0	?	0.4	48.7	✔	0.7	18.5	$\langle 0, F, ls\rangle$
18. `order-irrel`$^+$	?	200.6	54.2	$\langle cs{:}1, tp{:}F, pg\rangle$	☢	0.1	20.1	?	14.0	256.8	✔	6.3	16.9	$\langle 1, T, pg\rangle$
19. `order-irrel-nondet`	?	193.8	167.6	$\langle cs{:}1, tp{:}F, ls\rangle$	☢	0.1	20.0	**T**	575.2	1373.2	✔	4.4	22.4	$\langle 1, T, ls\rangle$
20. `overview1`	✔	11.5	29.2	$\langle cs{:}1, tp{:}F, ls\rangle$	✔	3.2	221.5	✔	49.2	336.8	✔	0.4	13.1	$\langle 0, F, ls\rangle$
21. `reentr`	✔	9.6	21.2	$\langle cs{:}1, tp{:}F, ls\rangle$	**M**	46.4	1000.0	**T**	228.7	648.3	✔	0.4	11.0	$\langle 1, F, ls\rangle$
22. `resource-analysis`	✔	6.8	24.0	$\langle cs{:}1, tp{:}F, ls\rangle$	**T**	931.0	133.6	✔	3.8	140.8	✔	0.4	10.2	$\langle 1, F, ls\rangle$
23. `sum-appendix`	✔	8.2	27.9	$\langle cs{:}1, tp{:}F, ls\rangle$	☢	0.0	19.3	?	0.4	48.0	✔	0.0	6.1	$\langle 1, T, ls\rangle$
24. `sum-of-ev-even`$^+$	✔	13.5	17.6	$\langle cs{:}0, tp{:}F, pg\rangle$	☢	0.0	19.0	?	0.8	122.8	✔	0.5	10.5	$\langle 0, F, pg\rangle$
25. `temperature`$^+$	?	287.6	625.9	$\langle cs{:}1, tp{:}F, oct\rangle$	☢	0.1	21.9	?	182.9	534.3	✔	35.3	26.1	$\langle 1, F, pg\rangle$
geomean for ✔*'s:*		**7.5**				**0.6**			**11.2**			**1.9**		

In addition to the above, we also provide 25 unsafe *benchmarks.*
evDrift *analyzed all of them (using the Config in the last column above) in 273s.*

Table 1. Comparison of **ev**Drift against assertion verifiers for effect-free programs: Drift, RCaml/PCSat and MoCHi via our tuple reduction. Benchmarks marked with $^+$ involve the `mod` operation. 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. **ev**Drift **verified additional 12, 20** and **15** than Drift, RCaml/PCSat and MoCHi, respectively, could not, and it is **25× faster** than Drift on Drift-verifiable examples, **3.4× faster** than RCaml/PCSat on RCaml/PCSat-verifiable examples, and **16× faster** on MoCHi-verifiable examples.

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 verify several benchmarks like `disj`, `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` and `disj` that have if-then-else statements, without using trace partitioning.

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. It also fails to verify certain higher-order programs like `market`. **ev**Drift, like Drift, is able to infer rich relationships between input and output types for program variables. However, since accumulator variables are abstracted in a statewise manner, it is much harder to form such relationships for their input and output effects. We believe that this can be improved in the future by more precisely tracking states from input to output effects.

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 Apx. G. In summary, **ev**Drift is able to verify two more benchmarks with trace partitioning, yet it slows down by 0.8×.

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 25× on DRIFT-verifiable programs), as compared with prior effect-less tools (DRIFT, RCAML/PCSAT and MoCHI) which require translations to encode effects.

Other related work. We discuss 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. [33] 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 are inspired by their work and build on aspects of their type system, abstract interpretation and implementation. Also related are other type and effect systems [15, 24, 32], however their treatments of effects are not accumulative, which is fundamental to our abstract effect domain. Moreover, those works do not provide an implementation. Hofmann and Chen [15] 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. [31] described a method of verifying temporal properties of higher-order programs through the Vardi [45] 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 infinite execution behaviors while we only verify finite execution properties and (ii) we support expressive SAA properties of finite traces, 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 [20, 25, 37]. Kobayashi's [22, 23] 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. 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. Lago and Ghyselen [27] 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. [20] 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. [24]), as well as a combination of both least and greatest fixpoint reasoning for abstract interpretations.

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. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/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. https://doi.org/10.1145/512760.512770

[8] 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. https://doi.org/10.1007/978-3-030-25540-4_1

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

[10] 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. https://doi.org/10.1145/113445.113468

[11] 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. https://doi.org/10.1007/978-3-642-39799-8_2

[12] hiroshi unno. [n. d.]. CoAR. https://github.com/hiroshi-unno/coar/

[13] 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. https://doi.org/10.1145/1926385.1926427

[14] 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. https://doi.org/10.1145/3009837.3009842

[15] 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. https://doi.org/10.1145/2603088.2603127

[16] hopv. [n. d.]. MoCHi. https://github.com/hopv/mochi

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

[18] 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. https://doi.org/10.1007/978-3-642-02658-4_52

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

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

[21] 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. https://doi.org/10.1145/1481848.1481853

[22] 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. https://doi.org/10.1145/1480881.1480933

[23] 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. https://doi.org/10.1145/1993498.1993525

[24] 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. https://doi.org/10.1145/2603088.2603138

[25] 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

[26] 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. https://doi.org/10.1007/978-3-642-54833-8_21

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

[28] 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. https://doi.org/10.1145/73560.73564

[29] 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. https://doi.org/10.1007/978-3-540-31987-0_2

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

[31] 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. https://doi.org/10.1145/2837614.2837667

[32] 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. https://doi.org/10.1145/3209108.3209204

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

[34] 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. https://doi.org/10.1109/SFCS.1977.32

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

[36] 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. https://doi.org/10.1145/1375581.1375602

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

//doi.org/10.1145/3571264

[38] 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. https://doi.org/10.1145/3009837.3009885

[39] 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. https://doi.org/10.1007/978-3-540-30477-7_8

[40] 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. https://doi.org/10.1017/S0956796807006466

[41] sosy lab. [n. d.]. benchexec. https://github.com/sosy-lab/benchexec

[42] 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. https://doi.org/10.1109/SP40001.2021.00085

[43] 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. https://doi.org/10.1145/1706299.1706315

[44] ucsd progsys. [n. d.]. LiquidHaskell. https://ucsd-progsys.github.io/liquidhaskell/

[45] 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.

[46] 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. https://doi.org/10.1145/2633357.2633366

[47] 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. https://doi.org/10.1145/292540.292560