A HAT Trick: Automatically Verifying Representation Invariants Using Symbolic Finite Automata

ANONYMOUS AUTHOR(S)

Functional programs typically interact with effectful libraries that hide state behind typed abstractions. One particularly important class of applications are data structure implementations that rely on such libraries to provide a level of efficiency and scalability that may be otherwise difficult to achieve. However, because the specifications of the methods provided by these libraries are necessarily general and rarely specialized to the needs of any specific client, any required application-level invariants must often be expressed in terms of additional constraints on the (often) opaque state maintained by the library.

In this paper, we consider the specification and verification of such *representation invariants* using *symbolic finite automata* (SFA). We show that SFAs can be used to succinctly and precisely capture fine-grained temporal and data-dependent histories of interactions between functional clients and effectful libraries. To facilitate modular and compositional reasoning, we integrate SFAs into a refinement type system to qualify effectful computations resulting from such interactions. The particular instantiation we consider, *Hoare Automata Types* (HATs), allows us to both specify and automatically type-check the representation invariants of a datatype, even when its implementation depends on effectful library methods that operate over hidden state.

We also develop a new bidirectional type checking algorithm that implements an efficient subtyping inclusion check over HATs, enabling their translation into a form amenable for SMT-based automated verification. We present extensive experimental results on an implementation of this algorithm that demonstrates the feasibility of type-checking complex and sophisticated HAT-specified OCaml data structure implementations layered on top of effectful library APIs.

1 INTRODUCTION

Functional programs often interact with effectful libraries that hide internal state behind their APIs. Such libraries are a vital component of a programming language's ecosystem, providing well-tested implementations of databases, key-value stores, logging and system services, and other effectful abstractions. A common use case for these effectful libraries is for programmers to build higher-level data structures on top of the abstractions they provide [5]. A set abstract data type (ADT) equipped with intersection and cardinality operations, for example, might use the aforementioned key-value library to maintain a persistent collection of its elements by associating those elements (represented as values) with unique ids (their associated keys). Since the internal state of the underlying effectful library is opaque to its clients, the implementations of these higher-level modules will necessarily be expressed in terms of the operations provided by the library: the methods of the set ADT might use methods like **put** and **get** to interact with the backing key-value store, for example. To maximize reusability, these low-level libraries are typically quite permissive in how client may use them, resulting in weak specifications that describe simple algebraic or equational constraints. For example, the specification of the key-value store might only state that two **put** operations over different keys always commute or that a **get** operation on a key always returns the last value **put** in that key.

In contrast, the higher-level abstractions provided by clients built on top of these libraries are equipped with stronger guarantees: the operations of our set ADT need to respect the semantics of mathematical sets, for example, so that $\forall A\ B : \texttt{Set.t.}\ |A \cup B| + |A \cap B| = |A| + |B|$. Verifying such semantically rich properties typically requires a strong guarantee that the internal state of the set library is consistent with the abstract states of the datatype, i.e., a *representation invariant* [26, 27]. Ensuring that distinct keys are always associated with distinct values allows us to prove the above

2023. XXXX-XXXX/2023/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

relationship between the intersection and cardinality operations of the set ADT, for example. Establishing such an invariant is complicated by the fact that our set ADT is built on top of another library with opaque internal state, and whose weak specifications do not directly address notions of uniqueness central to the set ADT. This paper presents an automated type-based verification procedure that addresses these complications.

Our proposed solution augments a refinement type system for a core call-by-value functional core language that supports calls to stateful libraries, with *Hoare Automata Types* (HATs). HATs are a new type abstraction that qualifies basic types with pre- and post-conditions expressed as *symbolic finite-state automata* [42, 43], a generalization of finite-state automata equipped with an unbounded alphabet, and whose transitions are augmented with predicates over the elements of this alphabet. HATs serve as a compact representation of the trace of effectful API calls and returns made prior to a function invocation and those performed during its execution. As a result, HATs serve as an expressive basis for a fine-grained effect system within a general refinement type framework, enabling compositional reasoning about invocations of stateful library APIs in a form amenable to automated verification of a client module's representation invariants. The representation invariant from our set ADT, for example, is captured by the following symbolic automaton, written using a variant of linear temporal logic on finite sequences [8]. The base predicates of this formula describe *events*, or invocations of the methods of a key-value store whose values are drawn from a `setElem` type:

$$I_{\texttt{Set}}(\texttt{el}) \doteq \Box(\langle \textbf{put}\ key\ val = v \mid val = \texttt{el}\rangle \implies \bigcirc\neg\Diamond\langle \textbf{put}\ key\ val = v \mid val = \texttt{el}\rangle)$$

This representation invariant ($I_{\texttt{Set}}$) uses the standard temporal logic modalities (e.g., $\Box$ (always), $\bigcirc$ (next), and $\Diamond$ (eventually)) to express the history of valid interactions between the set implementation and the underlying key-value store that ensure the expected element uniqueness property holds. Informally, the invariant establishes that, at every step of the execution ($\Box(...)$), once a value `el` has been inserted into the set ($\langle \textbf{put}\ key\ val = v \mid val = \texttt{el}\rangle$), it will never be reinserted ($\bigcirc\neg\Diamond\langle \textbf{put}\ key\ val = v \mid val = \texttt{el}\rangle$). An implementation that preserves this invariant guarantees that every value is always associated with at most one key.

To do so, we can check the `insert` method of the set ADT against the following type

$$\tau_{\texttt{insert}} \doteq \texttt{el}{:}\texttt{setElem} \dashrightarrow \texttt{x}{:}\{v{:}\texttt{setElem} \mid \top\} \rightarrow [I_{\texttt{Set}}(\texttt{el})]\,\{v{:}\texttt{unit} \mid \top\}\,[I_{\texttt{Set}}(\texttt{el})]$$

The output type of $\tau_{\texttt{insert}}$ ($[I_{\texttt{Set}}(\texttt{el})]\,\{v{:}\texttt{unit} \mid \top\}\,[I_{\texttt{Set}}(\texttt{el})]$) is a Hoare Automata Type (HAT) over the representation invariant $I_{\texttt{Set}}$. This type captures the behavior required of any correct implementation of `insert`: assuming the invariant holds for any value `el` (encoded here as a ghost variable scoped using $\dashrightarrow$) prior to the insertion of a new element `x`, it must also hold after the insertion as well. The precondition of the HAT captures the set of interactions that lead to a valid underlying representation state (namely, one in which every value in the store is unique), and the postcondition of the HAT specifies that this condition remains valid even after executing `insert`.

Our use of HATs differs in important ways from other type-based verification approaches for functional programs that allow computations to perform fine-grained effects, such as F* [38] or Ynot [29]. These systems use the power of rich dependent types to specify fine-grained changes to the heap induced by an effectful operation. The Dijkstra monad [25, 39] used by F*, for example, extends ideas first proposed by Nanevski *et. al* [30] in their development of Hoare Type Theory to embed a weakest precondition predicate transformer semantics into a type system that can be used to generate verification conditions consistent with a provided post-condition on the program heap. In contrast, our setting assumes that effectful computations are executed by a library with a *hidden* state, requiring the representation invariant of clients who use the library to be expressed solely in terms of observations on calls (and returns) to (and from) the methods of the underlying library.

Our approach also bears some similarity to recent (refinement) type and effect systems for verifying temporal properties of event traces *produced* during a computation [31, 35]. While the idea of using symbolic traces of previous observations as a handle to reason about the hidden state of a library is similar to these other efforts, incorporating return values of operations as part of such traces is a significant point of difference. This capability, critical for the validation of useful representation invariants, allows the precondition encoded in a HAT to describe a fine-grained *context* of events in which a program can run, expressed in terms of both library method arguments *and* return values. These important differences with prior work position HATs as a novel middle-ground in the design space between explicit state-based verification techniques and trace-based effect systems. Our formulation also leads to a number of significant technical differences with other refinement type systems [21] and their associated typing algorithms, since we must now contend with type-level reasoning over symbolic automata qualifiers.

In summary, this paper makes the following contributions:

(1) We formalize a new trace-based specification framework for expressing representation invariants of functional clients of effectful libraries that manage hidden state.
(2) We show how this framework can be embedded within a compositional and fine-grained effect-tracking refinement type system, by encoding specifications as symbolic automata and instantiated as HATs in the type system.
(3) We develop a bidirectional type-checking algorithm that translates the declarative type system into efficient subtype inclusion checks amenable to SMT-based automated verification.
(4) Using a tool (Marple) that implements these ideas, we present a detailed evaluation study over a diverse set of non-trivial OCaml datatype implementations that interact with stateful libraries. To the best of our knowledge, Marple is the first system capable of automated verification of sophisticated representation invariants for realistic OCaml programs.

The remainder of this paper is organized as follows. In the next section, we provide further motivation for the problem by developing a running example, which is used in the rest of the paper. Sec. 3 introduces a core functional language equipped with operators that model effectful library methods. Sec. 4 presents a refinement type system that augments basic type qualifiers to also include HATs; a brief background on symbolic automata is also given. An efficient bidirectional typing algorithm is then presented in Sec. 5. Sec. 6 provides implementation details of Marple, as well as experimental results on its effectiveness in automatically verifying the representation invariants of a number of realistic data structure client programs that interact with effectful libraries. The paper ends with a discussion of related work and conclusions.

2 MOTIVATION

To further motive our approach, consider the (highly-simplified) Unix-like file-system directory abstraction (`FileSystem`) shown in Fig. 1. This module is built on top of other libraries that maintain byte arrays (`Bytes.t`) which hold a file or a directory's contents and metadata, and which manipulate *paths* (`Path.t`), strings that represent fully-elaborated file and directory names. `FileSystem`'s implementation also uses an underlying key-value store library (`KVStore`) to provide a persistent and scalable storage abstraction, where keys represent file paths and values are byte arrays. The interface to this store is defined by three *effectful* library methods: **put** persistently stores a key-value pair $\langle k, v \rangle$, adding the pair to the store if k does not already exist, and overwriting its old value with v if it does; **exists** returns true if the given key has been previously **put** and false otherwise; and, **get** returns the value associated with its key argument, and fails (i.e., raises an exception) if the key does not exist in the store.

Fig. 1 shows three of the methods provided by `FileSystem`: `init` initializes the file system by storing a root path (“`/`”); `add` allows users to add regular files or directories to the filesystem; and, `delete` logically removes files or directories from the filesystem. The latter two operations return a

```
module type Bytes = sig
  type t
  ...
end

module type Path = sig
  type t = string
  val parent: t -> t
  val isRoot: t -> bool
  ...
end

module type File = sig
  val init: unit -> Bytes.t
  val isDir: Bytes.t -> bool
  val isFile: Bytes.t -> bool
  val isDel: Bytes.t -> bool
  val setDeleted: Bytes.t -> Bytes.t
  val delChild:
    Bytes.t -> Path.t -> Bytes.t
  val addChild:
    Bytes.t -> Path.t -> Bytes.t
  ...
end

module type KVStore = sig
  val put: Path.t -> Bytes.t -> unit
  val exists: Path.t -> bool
  val get: Path.t -> Bytes.t
end
```

```
1  module FileSystem = struct
2    open KVStore
3    let init () : unit = put "/" (File.init ())
4
5    let add (path: Path.t) (bytes: Bytes.t): bool =
6      if exists path then false
7      else
8        let parent_path = Path.parent path in
9        if not (exists parent_path) then false
10       else
11         let bytes' = get parent_path in
12         if File.isDir bytes' then (
13           put path bytes;
14           put parent_path (File.addChild bytes' path);
15           true)
16         else false
17
18   let delete (path : Path.t): bool =
19     if (isRoot path) or not (exists path) then false
20     else (
21       let bytes = get path in
22       if File.isDir bytes then deleteChildren(path);
23       let parent_path = Path.parent path in
24       let bytes' = get parent_path in
25       put path (File.setDeleted bytes);
26       put parent_path (File.delChild bytes' path);
27       true)
28 end
```

Fig. 1. A file system datatype based on underline key-value store library.

Boolean value to indicate whether they were successful. The add operation succeeds (line 15) only if (a) the given path does not already exist in the file system (line 6) and (b) a path representing the parent directory for the file does exist (line 9). Besides adding the file (line 13), add also updates the contents of the parent directory to make it aware that a new child has been added (line 14). Similar to add, `delete` requires that the supplied path exists and is not the root (line 19). If the path corresponds to a directory, its children are marked deleted (line 22) using procedure `deleteChildren` (not shown). Additionally, the directory's metadata, and that of its parent, are updated to record the deletion (lines 25 and 26). Implicit in `delete`'s implementation is the assumption that the parent path of the file to be deleted exists in the store: observe that there is no check to validate the existence of this path before the **get** of the parent's contents is performed on line 24. This assumption can be framed as a representation invariant that must hold prior to and after every call and return of add and `delete`. Intuitively, this invariant captures the following property:

Invariant$_{FS}$: *Any path that is stored as a key in the key-value store (other than the root path) must also have its parent stored as a non-deleted directory in the store.*

Representation Invariants. In a language that guarantees proper data abstraction, library developers can safely assume that any guarantees about an API established using a representation invariant are independent of a particular client's usage. However, as we noted earlier, numerous challenges arise when validating a representation invariant in our setting because the state of the underlying library is opaque to the client, as establishing this invariant crucially depends on precisely characterizing the admissible sequence of interactions between the client datatype and the backing library. In our example, this characterization must be able to capture the structure of a valid filesystem, in terms sequences of **put**, **get**, and **exists** calls and returns that are sufficient to ensure that the invariant is preserved. Verifying that $\textbf{Invariant}_{FS}$ holds requires us to prove that, except for the root, the path associated with a file or directory can only be recorded in the store when its parent path exists, i.e., its parent has previously been **put** and has not yet been deleted. Establishing that this property holds involves reasoning over both data dependencies among filesystem calls and temporal dependencies among store operations.

Events, Traces, and Symbolic Finite Automata. We address these challenges by encoding desired sets of effectful interactions between the client and library as *symbolic finite automata* (SFA). Besides constants and program variables, the alphabet of an automaton that captures these interactions includes a set of effectful operators **op** (e.g., **put** and **get**). The language recognized by an automaton dictates the sequences of permitted library method invocations. Each such invocation (called an *event*) $\textbf{op}\,\overline{v_i} = v$ records the operator invoked (**op**), a list of its arguments ($\overline{v_i}$), and a single result value v. It is critical that we record *both* inputs and outputs of an effectful operation, in order to enable our type system to distinguish among different hidden library states. Consider, for example, an effectful library initialization operator **initialize** : `unit` → `bool`. The hidden state after observing "**initialize** () = `true`", representing a successful initialization action, is likely to be different from the hidden state that exists after observing the event "**initialize** () = `false`", which indicates an initialization error. The language accepted by an SFA is a potentially infinite set of finite sequences of events, or *traces*.

Example 2.1 (Traces). In contrast to the correct implementation of `add` shown in Fig. 1, consider the following incorrect version:

```
let add_bad (path: Path.t) (bytes: Bytes.t) = put path bytes
```

This implementation simply records a path without considering whether its parent exists. We assume the file system initially contains only the root path, "/", which is **put** when the file system is created by the method `init`. The traces α_1 and α_2, shown below, represent executions of these two implementations that reflect their differences:

$$\begin{aligned}
&\text{under context trace } \alpha_0 \doteq [\textbf{put}\,\text{“/”}\,\texttt{bytesDir} = ()] \\
\texttt{add}_{\texttt{bad}}\,\text{“}\texttt{/a/b.txt}\text{”}\,\texttt{bytesFile}\ &\text{yields } \alpha_1 \doteq [\textbf{put}\,\text{“/”}\,\texttt{bytesDir} = ();\ \textbf{put}\,\text{“}\texttt{/a/b.txt}\text{”}\,\texttt{bytesFile} = ()] \\
\texttt{add}\,\text{“}\texttt{/a/b.txt}\text{”}\,\texttt{bytesFile}\ &\text{yields } \alpha_2 \doteq [\textbf{put}\,\text{“/”}\,\texttt{bytesDir} = ();\ \textbf{exists}\,\text{“}\texttt{/a/b.txt}\text{”} = \texttt{false};\ \textbf{exists}\,\text{“}\texttt{/a}\text{”} = \texttt{false}]
\end{aligned}$$

where `bytesDir` and `bytesFile` represent the contents of a directory and a regular file, resp. The trace α_1 violates $\textbf{Invariant}_{FS}$ since it registers the path "`/a/b.txt`" in the file system even though the parent path ("`/a`") does not exist. Thus, performing the operation `delete` "`/a/b.txt`" after executing the actions in this trace will cause a runtime error when it attempts to perform a **get** operation on the parent path "`/a`". On the other hand, executing this operation after executing the actions in trace α_2 terminates normally and preserves $\textbf{Invariant}_{FS}$.

Example 2.2 (Symbolic Finite Automata). We express symbolic automata as formulae in a symbolic version of linear temporal logic on finite sequences [8]. $\textbf{Invariant}_{FS}$ can be precisely expressed in this language as the following formula (denoted $I_{\mathsf{FS}}(\mathsf{p})$) parameterized over a path p:

$$I_{\mathsf{FS}}(\mathsf{p}) \doteq \Box\langle \mathsf{isRoot}(\mathsf{p})\rangle \lor (P_{\mathsf{isFile}}(\mathsf{p}) \lor P_{\mathsf{isDir}}(\mathsf{p}) \implies P_{\mathsf{isDir}}(\mathsf{parent}(\mathsf{p})))$$

where

$$P_{\mathsf{isDir}}(\mathsf{p}) \doteq \Diamond(\langle \textbf{put}\,\mathit{key}\,\mathit{val} = v \mid \mathit{key} = \mathsf{p} \land \mathsf{isDir}(\mathit{val})\rangle \land \bigcirc\Box\neg\langle \textbf{put}\,\mathit{key}\,\mathit{val} = v \mid \mathit{key} = \mathsf{p} \land (\mathsf{isDel}(\mathit{val}) \lor \mathsf{isFile}(\mathit{val}))\rangle)$$
$$P_{\mathsf{isFile}}(\mathsf{p}) \doteq \Diamond(\langle \textbf{put}\,\mathit{key}\,\mathit{val} = v \mid \mathit{key} = \mathsf{p} \land \mathsf{isFile}(\mathit{val})\rangle \land \bigcirc\Box\neg\langle \textbf{put}\,\mathit{key}\,\mathit{val} = v \mid \mathit{key} = \mathsf{p} \land (\mathsf{isDel}(\mathit{val}) \lor \mathsf{isDir}(\mathit{val}))\rangle)$$

Here, isRoot, isDir, isFile, and parent are pure functions on paths and bytes. The invariant is comprised of two disjunctions involving p:

(1) $\Box\langle \texttt{isRoot}(\texttt{p})\rangle$ asserts that p is the filesystem root and is always a valid path, regardless of any other actions that have been performed on the underlying key-value store.
(2) $P_{\texttt{isFile}}(\texttt{p}) \vee P_{\texttt{isDir}}(\texttt{p}) \implies P_{\texttt{isDir}}(\texttt{parent}(\texttt{p}))$ asserts that if p corresponds to either a file or directory in the file system, then its parent path must also correspond to a directory in the file system. The predicate $P_{\texttt{isDir}}(\texttt{p})$ captures contexts in which p has been registered as a directory and then not subsequently deleted or replaced by a file (i.e., $\bigcirc\Box\neg\langle \textbf{put}\; key\; val = v \mid key = \texttt{p} \wedge (\texttt{isDel}(val) \vee \texttt{isFile}(val))\rangle$). The predicate ($P_{\texttt{isFile}}(\texttt{p})$) is defined similarly except that it requires that an existing file not be deleted or modified to become a directory. Observe that atomic predicates (e.g., $\langle \textbf{put}\; key\; val = v \mid key = \texttt{p} \wedge \texttt{isFile}(val)\rangle$) not only capture when an event has been induced by an effectful operator (e.g., **put**), but also *qualify* the arguments to that operator and the result it produced (e.g., the argument *key* must be equal to the path p).[1]

This specification of the representation invariant $I_{\textsf{FS}}(\texttt{p})$ thus formalizes our informal characterization of a correct file system, $\textbf{Invariant}_{FS}$. The two subformulas capture the property that any path recorded as a key in the key-value store, other than the root, must have a parent directory. Notably, this specification is defined purely by constraining the sequence of **put** operations allowed on the store and does not require any knowledge of the representation of the underlying store.

Hoare Automata Types. While SFAs provide a convenient language in which to encode representation invariants, it is not immediately apparent how they can help with verification since, by themselves, they do not impose any safety constraints on expressions. To connect SFAs to computations, we embed them as refinements in a refinement type (and effect) system. In a standard refinement type system, a refinement type $\{v{:}b \mid \phi\}$ uses a qualifier ϕ to refine the value of an expression e. Similarly, a Hoare Automata Type (HAT), $[A]\,\{v{:}b \mid \phi\}\,[B]$, additionally refines a pure refinement type $\{v{:}b \mid \phi\}$ with two symbolic automata: a precondition automaton A defining an effect *context* under which e is allowed to evaluate, and a postcondition automata B that augments this context with the effects induced by e as it executes. A HAT thus allows us to track the sequence of effectful interactions between higher-level datatypes and the underlying stateful library, as well as to record the consequences of this interaction in terms of the return values of effectful calls.

Our type system leverages this information to verify the preservation of stated representation invariants in terms of these interactions. For example, add and delete can be ascribed the following HAT-enriched types:

$$\texttt{add} : \texttt{p}{:}\texttt{Path.t} \dashrightarrow \texttt{path}{:}\{v{:}\texttt{Path.t} \mid \top\} \to \texttt{bytes}{:}\{v{:}\texttt{Bytes.t} \mid \top\} \to [I_{\textsf{FS}}(\texttt{p})]\,\{v{:}\texttt{bool} \mid \top\}\,[I_{\textsf{FS}}(\texttt{p})] \qquad (\tau_{\texttt{add}})$$
$$\texttt{delete} : \texttt{p}{:}\texttt{Path.t} \dashrightarrow \texttt{path}{:}\{v{:}\texttt{Path.t} \mid \top\} \to [I_{\textsf{FS}}(\texttt{p})]\,\{v{:}\texttt{bool} \mid \top\}\,[I_{\textsf{FS}}(\texttt{p})] \qquad (\tau_{\texttt{delete}})$$

Here, the notation "$\texttt{p}{:}\texttt{Path.t} \dashrightarrow$" indicates that p is a ghost variable representing an arbitrary path; thus, the representation invariant must hold over *any* instantiation of p. While their input arguments ($\{v{:}\texttt{Path.t} \mid \top\}$ and ($\{v{:}\texttt{Bytes.t} \mid \top\}$)) are unconstrained, the return types of these functions are expressed as HATs in which both the precondition and postcondition automata refer to the representation invariant for the file system, $I_{\textsf{FS}}(\texttt{p})$. Informally, this type reads "if we invoke the function in a context that is consistent with $I_{\textsf{FS}}$, then the state after the function returns should also be consistent with $I_{\textsf{FS}}$". The traces admitted by an SFA are used to represent these contexts and states.

[1]Throughout this paper, we italicize event arguments.

Variables		$x, y, z, f, \nu, \ldots$
Pure Operations	$op ::=$	$+ \mid - \mid == \mid < \mid \leq \mid \mathsf{mod} \mid \mathsf{parent} \mid \ldots$
Effectful Operations	$\mathbf{op} ::=$	$\mathbf{put} \mid \mathbf{exists} \mid \mathbf{get} \mid \mathbf{insert} \mid \mathbf{mem} \mid \ldots$
Data constructors	$d ::=$	$() \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{O} \mid \mathsf{S} \mid \ldots$
Constants	$c ::=$	$\mathbb{Z} \mid d(\overline{c})$
Values	$v ::=$	$c \mid x \mid d(\overline{v}) \mid \lambda x{:}t.e \mid \mathbf{fix} f{:}t.\lambda x{:}t.e$
Expressions (Computations)	$e ::=$	$v \mid \mathbf{let}\ x = \mathbf{op}\ \overline{v}\ \mathbf{in}\ e \mid \mathbf{let}\ x = op\ \overline{v}\ \mathbf{in}\ e \mid \mathbf{let}\ x = v\ v\ \mathbf{in}\ e$
		$\mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid \mathbf{match}\ v\ \mathbf{with}\ \overline{d\ \overline{y} \to e}$

Fig. 2. $\boldsymbol{\lambda}^E$ term syntax.

$$\textbf{Traces} \quad \alpha ::= \quad [\,] \mid (\mathbf{op}\,\overline{v} = v) :: \alpha$$

Small-Step Reduction Rules $\boxed{\alpha \vDash e \overset{\alpha}{\hookrightarrow} e}$

$$\frac{op\ \overline{v} \Downarrow v_x}{\alpha \vDash \mathbf{let}\ x = op\ \overline{v}\ \mathbf{in}\ e \overset{[\,]}{\hookrightarrow} e[x \mapsto v_x]}\ \textsc{StPureOp} \qquad \frac{\alpha \vDash \mathbf{op}\ \overline{v} \Downarrow v_x}{\alpha \vDash \mathbf{let}\ x = \mathbf{op}\ \overline{v}\ \mathbf{in}\ e \overset{[\mathbf{op}\ \overline{v} = v_x]}{\hookrightarrow} e[x \mapsto v_x]}\ \textsc{StEffOp}$$

Fig. 3. Trace syntax and selected operational semantics.

More concretely, observe that the trace α_2 from Example 2.1 is derived by concatenating α_0 with the library calls generated during the (symbolic) execution of add:

$$\alpha_{\mathsf{new}} \doteq [\mathbf{exists}\ \text{“/a/b.txt”} = \mathsf{false};\ \mathbf{exists}\ \text{“/a”} = \mathsf{false}]$$

(i.e., $\alpha_2 = \alpha_0 {+\!\!+} \alpha_{\mathsf{new}}$). Note that both α_0 and α_2 satisfy the representation invariant I_{FS}. To admit this trace, our type system automatically infers a new automata A that accepts the sequence of events in α_{new} by type-checking add. Verifying that the representation invariant holds now reduces to an inclusion check between two symbolic automata, requiring that the trace expected by the *concatenated* automaton $I_{\mathsf{FS}}; A$ is also accepted by the representation invariant I_{FS}.

3 LANGUAGE

To formalize our approach, we first introduce $\boldsymbol{\lambda}^E$, a core calculus for effectful programs.

Syntax. $\boldsymbol{\lambda}^E$ is a call-by-value lambda calculus with built-in inductive datatypes, pattern matching, a set of pure and effectful operators, and recursive functions. Its syntax is given in Fig. 2. For simplicity, programs are expressed in monadic normal-form (MNF) [18], a variant of A-Normal Form (ANF) [15] that permits nested let-bindings. Terms in $\boldsymbol{\lambda}^E$ are syntactically divided into two classes: values v and expressions e, which include both pure terms and those that have computational effects. Similar to the simply-typed lambda calculus, function parameters have explicit type annotations. $\boldsymbol{\lambda}^E$ is parameterized over two sets of primitive operators: pure operators (op) include basic arithmetic and logical operators, while examples of effectful operators ($\mathbf{op}$) include state manipulating operators (e.g., **put**/**exists**/**get** for interacting with a key-value store, or **insert**/**mem** for manipulating an effectful set library). We express sequencing of effectful computations $e_1; e_2$ in terms of let-bindings: $e_1; e_2 \doteq \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$, where x does not occur free in e_2. The application $e_1\ e_2$ is syntactic sugar for $\mathbf{let}\ x = e_1\ \mathbf{in}\ \mathbf{let}\ y = e_2\ \mathbf{in}\ \mathbf{let}\ z = x\ y\ \mathbf{in}\ z$.

Operational Semantics. Our programming model does not have access to the underlying implementation of effectful operators, so the semantics of $\boldsymbol{\lambda}^E$ constrains the behavior of impure operations in terms of the outputs they may produce. To do so, we structure its semantics in terms of *traces* that record the history of previous effectful operations and their results. The syntax of traces adopts standard list operations (i.e., cons :: and concatenation $+\!\!+$), as shown in Fig. 3. *Traces* are lists of *effect events* $\mathbf{op}\,\overline{v} = v$ that record the application of an effectful operator $\mathbf{op}$ to the arguments $\overline{v}$ as well as the resulting value v. The traces α_2 and α_1 from Example 2.1 record the events generated by correct and incorrect implementations of the `add` function of the `FileSystem` ADT, for example.

The operational semantics of $\boldsymbol{\lambda}^E$ is defined via a small-step reduction relation, $\alpha \vDash e \xhookrightarrow{\alpha'} e'$. This relation is read as "under an effect context (i.e., trace) α, the term e reduces to e' in one step and performs the effect α'", where α' is either empty $[\,]$ or holds a single event. The semantics is parameterized over two auxiliary relations, $e \Downarrow v$ and $\alpha \vDash e \Downarrow v$, used to evaluate pure and impure operations, respectively. Fig. 3 shows how the corresponding reduction rules use these relations.[2] The reduction rule for pure operators (STPureOp) holds in an arbitrary effect context (including the empty one), and produces no additional events. The rule for effectful operators (STEffOp), in contrast, can depend on the current context α, and records the result value of the effectful operator in the event it emits. The multi-step reduction relation $\alpha \vDash e \xhookrightarrow{\alpha'}{}^{*} e'$ defines the reflexive, transitive closure of the single-step relation; its trace α records all events generated during evaluation.

Example 3.1. We can define the semantics for the **put**, **exists**, and **get** operators found in a key-value store library and used in `FileSystem`, via the following rules:

$$\frac{}{\alpha \vDash \mathbf{put}\,k\,v \Downarrow ()}\ \textsc{Put} \qquad \frac{\mathbf{put}\,k\,v = () \notin \alpha}{\alpha \vDash \mathbf{exists}\,k \Downarrow \mathtt{false}}\ \textsc{ExistsF} \qquad \frac{\mathbf{put}\,k\,v = () \in \alpha}{\alpha \vDash \mathbf{exists}\,k \Downarrow \mathtt{true}}\ \textsc{ExistsT} \qquad \frac{\mathbf{put}\,k\,v = () \notin \alpha'}{\alpha +\!\!+ [\mathbf{put}\,k\,v = ()] +\!\!+ \alpha' \vDash \mathbf{get}\,k \Downarrow v}\ \textsc{Get}$$

The rule Put asserts that **put** operations always succeed. For a particular key, the ExistsF and ExistsT rules stipulate that **exists** returns `false` or `true`, based on the presence of a corresponding **put** event in the effect context. Finally, the rule Get specifies that the result of a **get** event is the *last* value written by a **put** event to that key. Note that a program will get stuck if it attempts to **get** a key in a trace without an appropriate **put** event, since there is no rule covering this situation.

All the traces appearing in Sec. 2 can be derived from these rules. Under the effect context α_0, for example, the expression, `add` "`/a/b.txt`" `bytesFile`, induces the trace α_2 as follows: the `add` function uses the **exists** operator on the key "`/a/b.txt`" (line 6) to check if this path is in the file system. Since the context trace α_0 lacks a **put** event with the key "`/a/b.txt`", we have

$$[\mathbf{put}\,\text{“/”}\,\mathtt{bytesDir} = ()] \vDash \mathbf{exists}\,\text{“/a/b.txt”} \Downarrow \mathtt{false} \quad (\textsc{existsF})$$

The trace used to evaluate subsequent expressions records this event: $\alpha_0 +\!\!+ [\mathbf{exists}\,\text{“/a/b.txt”} = \mathtt{false}]$. The remaining evaluation of `add` "`/a/b.txt`" `bytesFile` is similar, yielding the following execution:

$$[\mathbf{put}\,\text{“/”}\,\mathtt{bytesDir} = ()] \vDash \mathtt{add}\,\text{“/a/b.txt”}\,\mathtt{bytesFile} \xhookrightarrow{[\mathbf{exists}\,\text{“/a/b.txt”} = \mathtt{false};\,\mathbf{exists}\,\text{“/a”} = \mathtt{false}]}{}^{*}\ \mathtt{false}$$

4 TYPE SYSTEM

The syntax of $\boldsymbol{\lambda}^E$'s types is shown in Fig. 4. Our type system uses *pure* refinement types to describe pure computations, and *Hoare Automata Types* (HATs) to describe effectful computations. Our pure refinement types are similar to those of other refinement type systems [21], and allow base types (e.g., `int`) to be further constrained by a logical formula or *qualifier*. Qualifiers in $\boldsymbol{\lambda}^E$ are universally quantified formulae over linear arithmetic operations (op) as well as uninterpreted functions, or *method predicates* (mp), e.g., `isRoot`. Verification conditions generated by our type-checking algorithm can nonetheless be encoded as effectively propositional (EPR) sentences, which

[2]The remaining rules are completely standard and provided in the supplemental material.

Type Syntax

$$
\begin{array}{rrcl}
\textbf{Base Types} & b & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \ldots \\
\textbf{Basic Types} & s & ::= & b \mid s \to s \\
\textbf{Value Literals} & l & ::= & c \mid x \mid op\ \overline{l} \mid mp\ \overline{l} \\
\textbf{Qualifiers} & \phi & ::= & l \mid \bot \mid \top \mid \neg\phi \mid \phi \land \phi \mid \phi \lor \phi \mid \phi \implies \phi \mid \forall x{:}b.\,\phi \\
\textbf{Refinement Types} & t & ::= & \{v{:}b \mid \phi\} \mid x{:}t \to \tau \mid x{:}b \dashrightarrow t \\
\textbf{Symbolic Finite Automata} & A, B & ::= & \langle \mathbf{op}\ \overline{x} = v \mid \phi \rangle \mid \langle \phi \rangle \mid \neg A \mid A \land A \mid A \lor A \mid A;A \mid \bigcirc A \mid A\ \mathcal{U}\ A \\
\textbf{Hoare Automata Types} & \tau & ::= & [A]\ t\ [A] \mid \tau \sqcap \tau \\
\textbf{Type Contexts} & \Gamma & ::= & \emptyset \mid x{:}t, \Gamma
\end{array}
$$

Type Aliases

$$\langle \mathbf{op} \ldots {\sim}v_x \ldots = v \mid \phi \rangle \doteq \langle \mathbf{op} \ldots x \ldots = v \mid x = v_x \land \phi \rangle \quad \Diamond A \doteq \langle \top \rangle\ \mathcal{U}\ A \quad \Box A \doteq \neg\Diamond\neg A \quad \mathbf{LAST} \doteq \neg\bigcirc\langle\top\rangle \quad b \doteq \{v{:}b \mid \top\}$$

Fig. 4. $\boldsymbol{\lambda}^E$ types.

can be efficiently handled by an off-the-shelf theorem prover like Z3 [9]. We also allow function types to be prefixed with a set of *ghost variables* with base types ($\overline{x{:}b} \dashrightarrow \tau$). Ghost variables are purely logical– they can only appear in other qualifiers and are implicitly instantiated when the function is applied.

Unique to $\boldsymbol{\lambda}^E$ are the HATs ascribed to an effectful computation, which constrain the traces it may produce. HATs use *Symbolic Finite Automata* (SFAs) [7, 12, 42] to qualify traces, similar to how standard refinement types use formulae to qualify the types of pure terms. We adopt the symbolic version of *linear temporal logic on finite traces* [8] as the syntax of SFAs.[3]

As shown in Fig. 4, HATs support two kinds of atomic propositions, or *symbolic effect events*: $\langle \mathbf{op}\ \overline{x} = v \mid \phi \rangle$ and $\langle \phi \rangle$. A symbolic effect event $\langle \mathbf{op}\ \overline{x} = v \mid \phi \rangle$ describes an application of an effectful operator **op** to a set of argument variables $\overline{x}$ that produces a result variable v. The formula ϕ constrains the possible instantiation of $\overline{x}$ and v in a concrete event. The other symbolic effect event, $\langle \phi \rangle$, is used to constrain the relationship between pure values (e.g., ghost variables and function arguments). In addition to the temporal next ($\bigcirc A$) and until operators ($A\ \mathcal{U}\ A$), the syntax of HATs includes negation ($\neg A$), intersection ($A \land A$), union ($A \lor A$), and concatenation ($A;A$) operators. Fig. 4 defines notations for other useful logical and temporal operators: implication ($A \implies B \doteq \neg A \lor B$), the eventually operator $\Diamond$, the globally operator $\Box$, and importantly the last modality **LAST**, which describes a singleton trace, thus prohibiting a trace from including any other effects [8]. Fig. 4 also provides notations for specifying the value of an argument $\langle \mathbf{op}\ \overline{x}\ \ {\sim}v_x\ \overline{x} = v \mid \phi \rangle$. As is standard, we abbreviate the qualified type $\{v{:}b \mid \top\}$ as b.

Example 4.1. This syntax can capture rich properties over effect contexts. For example, the following formulae concisely specify traces under which a key holds a certain value ($P_{\texttt{stored}}$) and when a particular key exists in the current store ($P_{\texttt{exists}}$):

$$P_{\texttt{stored}}(\texttt{key}, \texttt{value}) \doteq \Diamond(\langle \mathbf{put}\ {\sim}\texttt{key}\ {\sim}\texttt{value} = v \mid \top \rangle \land \bigcirc\Box\neg\langle \mathbf{put}\ {\sim}\texttt{key}\ x = v \mid \top \rangle) \quad P_{\texttt{exists}}(\texttt{key}) \doteq \Diamond\langle \mathbf{put}\ {\sim}\texttt{key}\ x = v \mid \top \rangle$$

4.1 HATs, By Example

Formally, a Hoare Automata Type $[A]\ t\ [B]$ qualifies a refinement type t with two SFAs: a *precondition* SFA A that captures the context traces in which a term can be executed and a *postcondition*

[3]Our type system is agnostic to the syntax used to express SFAs, and our implementation uses the more developer-friendly syntax of Symbolic Regular Languages [8]; this syntax is provided in supplemental materials.

SFA B that describes the effect trace after the execution of the term. Our type system also includes intersection types on HATs ($\tau \sqcap \tau$), in order to precisely specify the behaviors of a program under different effect contexts.

Example 4.2 (Built-in Effectful Operators). Our type system is parameterized over a (global) typing context Δ containing the types of built-in operators. Intuitively, the signatures of effectful operators in Δ captures the semantics the library developer ascribes to their API. Using the SFAs from Example 4.1, for example, the signatures of **put**, **exists**, and **get** are:

$$\Delta(\textbf{put}) = \mathtt{k{:}Path.t} \rightarrow \mathtt{a{:}Bytes.t} \rightarrow [\Box\langle\top\rangle]\,\mathtt{unit}\,[\Box\langle\top\rangle; (\langle \textbf{put}\ \sim\mathtt{k}\ \sim\mathtt{a} = \nu \mid \top\rangle) \land \textbf{LAST}]$$

$$\Delta(\textbf{get}) = \mathtt{a{:}Bytes.t} \dashrightarrow \mathtt{k{:}Path.t} \rightarrow [P_{\mathtt{stored}}(\mathtt{k},\mathtt{a})]\,\{\nu{:}\mathtt{Bytes.t} \mid \nu = \mathtt{a}\}\,[P_{\mathtt{stored}}(\mathtt{k},\mathtt{a}); (\langle \textbf{get}\ \sim\mathtt{k} = \nu \mid \nu = \mathtt{a}\rangle) \land \textbf{LAST}]$$

$$\begin{aligned}\Delta(\textbf{exists}) = \mathtt{k{:}Path.t} \rightarrow\ & [P_{\mathtt{exists}}(\mathtt{k})]\,\{\nu{:}\mathtt{bool} \mid \nu = \mathtt{true}\}\,[P_{\mathtt{exists}}(\mathtt{k}); (\langle \textbf{exists}\ \sim\mathtt{k} = \nu \mid \nu = \mathtt{true}\rangle) \land \textbf{LAST}] \sqcap \\ & [\neg P_{\mathtt{exists}}(\mathtt{k})]\,\{\nu{:}\mathtt{bool} \mid \nu = \mathtt{false}\}\,[\neg P_{\mathtt{exists}}(\mathtt{k}); (\langle \textbf{exists}\ \sim\mathtt{k} = \nu \mid \nu = \mathtt{false}\rangle) \land \textbf{LAST}]\end{aligned}$$

The postcondition SFAs of all three operators use **LAST**, to indicate that they append a single effect event to the end of the effect context ($\langle \textbf{op}\ \overline{v} = \nu \mid \phi\rangle \land \textbf{LAST}$). The type of **put** permits it to be used under any context ($\Box\langle\top\rangle$). More interestingly, the built-in type of **get** uses a ghost variable (a) to represent the current value of the key k; its precondition stipulates that it should only be applied to keys that have been previously stored. Finally, the intersection type of **exists** describes two possible behaviors, depending on whether the key k has been previously added to the store.

Example 4.3 (MinSet). Consider a developer who wants to implement an API for a set augmented with an additional operation that returns the minimum element in the set. This implementation is defined in terms of two other effectful libraries: a Set ADT that provides effectful operators **insert**:`int → unit` and **mem**:`int → bool`, and a MemCell library that provides a persistent cell with effectful **read**:`unit → int` and **write**:`int → unit` operators. One implementation strategy is to track the minimum element in a persistent cell, and maintain the representation invariant that the contents of this cell are always less than or equal to every element in the underlying set. Using SFAs, we can express this invariant as

$$\begin{aligned}I_{\mathtt{MinSet}}(\mathtt{el}) \doteq\ & \Diamond(\langle \textbf{write}\ \sim\mathtt{el} = \nu \mid \top\rangle \land \bigcirc\Box\neg\langle \textbf{write}\ x = \nu \mid \top\rangle) \implies \Diamond\langle \textbf{insert}\ \sim\mathtt{el} = \nu \mid \top\rangle \land \Box\neg\langle \textbf{insert}\ x = \nu \mid x < \mathtt{el}\rangle \\ & \land (\Box\neg\langle \textbf{write}\ x = \nu \mid \top\rangle \implies \Box\neg\langle \textbf{insert}\ x = \nu \mid \top\rangle)\end{aligned}$$

which specifies that an element `el` is written into the persistent cell only when `el` has been inserted into the backing Set *and* no element less than `el` has also inserted. The HAT ascribed to the function `minset_insert` enforces this invariant

$$\mathtt{el{:}int} \dashrightarrow \mathtt{elem{:}int} \rightarrow [I_{\mathtt{MinSet}}(\mathtt{el})]\,\mathtt{unit}\,[I_{\mathtt{MinSet}}(\mathtt{el})]$$

Example 4.4 (LazySet). As our next example, consider a LazySet ADT that provides a lazy version of an `insert` operation, delaying when elements are added to an underlying Set ADT. The ADT does so by returning a thunk closed over a collection of elements that will be added to the set when it is forced. This ADT maintains the representation invariant that an element is never inserted twice:

$$I_{\mathtt{LSet}}(\mathtt{el}) \doteq \Box(\langle \textbf{insert}\ \sim\mathtt{el} = \nu \mid \top\rangle \implies \bigcirc\neg\Diamond\langle \textbf{insert}\ \sim\mathtt{el} = \nu \mid \top\rangle)$$

We can specify a "*lazy*" insert operator using the following HAT:

$$\mathtt{el{:}int} \dashrightarrow \mathtt{elem{:}int} \rightarrow \mathtt{thunk{:}}(\mathtt{unit} \rightarrow [I_{\mathtt{LSet}}(\mathtt{el})]\,\mathtt{unit}\,[I_{\mathtt{LSet}}(\mathtt{el})]) \rightarrow \mathtt{unit} \rightarrow [I_{\mathtt{LSet}}(\mathtt{el})]\,\mathtt{unit}\,[I_{\mathtt{LSet}}(\mathtt{el})]$$

This operation takes as input a (potentially new) element `elem` to be added to the set and a thunk holding any elements that have not yet been inserted into the backing set, and returns another thunk. This HAT stipulates that both input thunk and output thunk preserve the representation invariant of the ADT: $\mathtt{unit} \rightarrow [I_{\mathtt{LSet}}(\mathtt{el})]\,\mathtt{unit}\,[I_{\mathtt{LSet}}(\mathtt{el})]$.

Example 4.5 (DFA). Our final example is a library built on top of an effectful graph library, that is used to maintain the states and transitions of a Deterministic Finite Automaton (DFA). The

Type Erasure $\boxed{\lfloor t \rfloor \quad \lfloor \tau \rfloor \quad \lfloor \Gamma \rfloor}$

$$\lfloor \{v{:}b \mid \phi\} \rfloor \doteq b \quad \lfloor x{:}t \to \tau \rfloor \doteq \lfloor t \rfloor \to \lfloor \tau \rfloor \quad \lfloor x{:}b \dashrightarrow t \rfloor \doteq \lfloor t \rfloor \quad \lfloor [A]\, t\, [B] \rfloor \doteq \lfloor t \rfloor \quad \lfloor \tau_1 \sqcap \tau_2 \rfloor \doteq \lfloor \tau_1 \rfloor$$
$$\lfloor \emptyset \rfloor \doteq \emptyset \qquad \lfloor x{:}t, \Gamma \rfloor \doteq x{:}\lfloor t \rfloor, \lfloor \Gamma \rfloor$$

Well-Formed Types $\boxed{\Gamma \vdash^{\mathbf{WF}} t \quad \Gamma \vdash^{\mathbf{WF}} A \quad \Gamma \vdash^{\mathbf{WF}} \tau}$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} A \quad \Gamma \vdash^{\mathbf{WF}} B \quad \Gamma \vdash^{\mathbf{WF}} t \\ \forall \sigma \in \llbracket \Gamma \rrbracket . \mathcal{L}(\sigma(B)) \subseteq \mathcal{L}(\sigma(A; \Box\langle \top \rangle))\end{array}}{\Gamma \vdash^{\mathbf{WF}} [A]\, t\, [B]} \textsc{WFHoare} \qquad \frac{\Gamma \vdash^{\mathbf{WF}} \tau_1 \quad \Gamma \vdash^{\mathbf{WF}} \tau_2 \quad \lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor}{\Gamma \vdash^{\mathbf{WF}} \tau_1 \sqcap \tau_2} \textsc{WFInter}$$

Automata Inclusion $\boxed{\Gamma \vdash A \subseteq A}$

$$\frac{\begin{array}{c}\forall \sigma \in \llbracket \Gamma \rrbracket . \\ \mathcal{L}(\sigma(A_1)) \subseteq \mathcal{L}(\sigma(A_2))\end{array}}{\Gamma \vdash A_1 \subseteq A_2} \textsc{SubAutomata}$$

Subtyping $\boxed{\Gamma \vdash t <: t \quad \Gamma \vdash \tau <: \tau}$

$$\frac{\begin{array}{c}\Gamma \vdash A_2 \subseteq A_1 \quad \Gamma \vdash t_1 <: t_2 \\ \Gamma \vdash (A_2; \Box\langle \top \rangle) \land B_1 \subseteq \\ (A_2; \Box\langle \top \rangle) \land B_2\end{array}}{\Gamma \vdash [A_1]\, t_1\, [B_1] <: [A_2]\, t_2\, [B_2]} \textsc{SubHoare}$$

$$\frac{}{\Gamma \vdash \tau_1 \sqcap \tau_2 <: \tau_1} \textsc{SubIntLL} \qquad \frac{}{\Gamma \vdash \tau_1 \sqcap \tau_2 <: \tau_2} \textsc{SubIntLR} \qquad \frac{}{\Gamma \vdash [A_1]\, t\, [B] \sqcap [A_2]\, t\, [B] <: [A_1 \lor A_2]\, t\, [B]} \textsc{SubIntMerge}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau <: \tau_1 \\ \Gamma \vdash \tau <: \tau_2\end{array}}{\Gamma \vdash \tau <: \tau_1 \sqcap \tau_2} \textsc{SubIntR} \qquad \frac{\Gamma, x{:}\{v{:}b \mid \top\} \vdash t_1 <: t_2}{\Gamma \vdash t_1 <: x{:}b \dashrightarrow t_2} \textsc{SubGhostR} \qquad \frac{\begin{array}{c}\lfloor \Gamma \rfloor \vdash_{\mathsf{S}} v : b \\ \Gamma \vdash t_1[x \mapsto v] <: t_2\end{array}}{\Gamma \vdash x{:}b \dashrightarrow t_1 <: t_2} \textsc{SubGhostL}$$

Fig. 5. Selected auxiliary typing relations. $\lfloor \Gamma \rfloor \vdash_{\mathsf{S}} e : \lfloor t \rfloor$ is the standard typing judgment for basic types.

underlying graph library exports two methods to add (**connect**) and remove (**disconnect**) edges. We can specify the standard invariant that DFAs only have deterministic transitions:

$$I_{\mathsf{DFA}}(\mathsf{n}, \mathsf{c}) \doteq \Box\neg(\langle \mathbf{connect} \sim\mathsf{n} \sim\mathsf{c}\ n_{end} = v \mid \top \rangle \land \bigcirc(\neg\langle \mathbf{disconnect} \sim\mathsf{n} \sim\mathsf{c}\ n_{end} = v \mid \top \rangle\ \mathcal{U}\ \langle \mathbf{connect} \sim\mathsf{n} \sim\mathsf{c}\ n_{end} = v \mid \top \rangle))$$

I_{DFA} stipulates that a node n can have at most one transition on a character c ($\langle \mathbf{connect} \sim\mathsf{n} \sim\mathsf{c}\ n_{end} = v \mid \top \rangle$). Adding a new transition from n on c requires that any previous transitions on c have first been removed ($\bigcirc(\neg\langle \mathbf{disconnect} \sim\mathsf{n} \sim\mathsf{c}\ n_{end} = v \mid \top \rangle\ \mathcal{U}\ \langle \mathbf{connect} \sim\mathsf{n} \sim\mathsf{c}\ n_{end} = v \mid \top \rangle)$). An `add_transition` method with the following signature is thus guaranteed to preserve determinism:

$$\mathtt{n{:}Node.t} \dashrightarrow \mathtt{c{:}Char.t} \dashrightarrow \mathtt{n\_start{:}Node.t} \to \mathtt{char{:}Char.t} \to \mathtt{n\_end{:}Node.t} \to [I_{\mathsf{DFA}}(\mathsf{n}, \mathsf{c})]\ \mathtt{unit}\ [I_{\mathsf{DFA}}(\mathsf{n}, \mathsf{c})]$$

4.2 Typing Rules

Type contexts. A type context (shown in Fig. 4) is a sequence of bindings from variables to pure refinement types (i.e., t). Type contexts are not allowed to contain HATs, as doing so breaks several structural properties (e.g., weakening) that are used to prove type safety.[4]

Auxiliary typing relations. Fig. 5 shows selected rules from three sets of auxiliary relations used by our type system. The first set describes the type erasure functions $\lfloor t \rfloor$, $\lfloor \tau \rfloor$, and $\lfloor \Gamma \rfloor$. The former two functions return basic types by erasing all qualifiers and automata from types, while the latter $\lfloor \Gamma \rfloor$ is the type context derived by applying $\lfloor ... \rfloor$ to all of Γ's bindings. The second set describes well-formedness conditions on SFAs ($\Gamma \vdash^{\mathbf{WF}} A$) and types ($\Gamma \vdash^{\mathbf{WF}} t$ and $\Gamma \vdash^{\mathbf{WF}} \tau$). These rules largely ensure that all the qualifiers appearing in a type are closed under the current typing context Γ.

[4]Intuitively, since a type of the form $[A]\, t\, [B]$ describes an effectful computation, it cannot be duplicated or eliminated, and thus it would be unsafe to allow unrestricted use of "computational" variables, as would be possible if type contexts could contain such bindings.

Typing $\boxed{\Gamma \vdash op : t \quad \Gamma \vdash \mathbf{op} : t \quad \Gamma \vdash e : t \quad \Gamma \vdash e : \tau}$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} A \\ \Gamma \vdash e : t\end{array}}{\Gamma \vdash e : [A]\, t\, [A]}\ \textsc{TEPur} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} \tau_2 \\ \Gamma \vdash \tau_1 <: \tau_2 \\ \Gamma \vdash e : \tau_1\end{array}}{\Gamma \vdash e : \tau_2}\ \textsc{TSub} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} \tau_1 \sqcap \tau_2 \\ \Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2\end{array}}{\Gamma \vdash e : \tau_1 \sqcap \tau_2}\ \textsc{TInter} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} x{:}b \dashrightarrow t \\ \Gamma, x{:}\{v{:}b \mid \top\} \vdash v : t\end{array}}{\Gamma \vdash v : x{:}b \dashrightarrow t}\ \textsc{TGhost}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} t \\ \Delta(op) = t\end{array}}{\Gamma \vdash op : t}\ \textsc{TPOp} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} t \\ \Delta(\mathbf{op}) = t\end{array}}{\Gamma \vdash \mathbf{op} : t}\ \textsc{TEOp}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} \tau \quad \Gamma \vdash op : \overline{z_i{:}t_i} \rightarrow t \\ \forall i. \Gamma \vdash v_i : t_i \\ t_x = t[z_i \mapsto v_i] \\ \Gamma, x{:}t_x \vdash e : \tau\end{array}}{\Gamma \vdash \mathbf{let}\ x = op\ \overline{v_i}\ \mathbf{in}\ e : \tau}\ \textsc{TPOpApp} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\mathbf{WF}} [A]\, t\, [B] \quad \Gamma \vdash \mathbf{op} : \overline{z_i{:}t_i} \rightarrow \tau \\ \forall i. \Gamma \vdash v_i : t_i \\ [A]\, t_x\, [A'] = \tau[z_i \mapsto v_i] \\ \Gamma, x{:}t_x \vdash e : [A']\, t\, [B]\end{array}}{\Gamma \vdash \mathbf{let}\ x = \mathbf{op}\ \overline{v_i}\ \mathbf{in}\ e : [A]\, t\, [B]}\ \textsc{TEOpApp}$$

Fig. 6. Selected typing rules. All typing judgements (i.e., $\Gamma \vdash e : t$ and $\Gamma \vdash e : \tau$) assume the corresponding basic type judgement ($\lfloor\Gamma\rfloor \vdash_{\mathrm{S}} e : \lfloor t \rfloor$ holds).

The exceptions are WFInter, which stipulates that only HATs with the same base type can be intersected, and WFHoare, which requires the language accepted by the precondition SFA to be a prefix of the postcondition automata (i.e., $\mathcal{L}(B) \subseteq \mathcal{L}(A; \Box\langle\top\rangle)$, where $\Box\langle\top\rangle$ denotes the SFA that accepts an arbitrary trace) after performing a substitution consistent with the current typing context ($\sigma \in [\![\Gamma]\!]$); the functions $\mathcal{L}(...)$ and $[\![...]\!]$ are defined in the next subsection.

Our type system also uses a mostly-standard subtyping relation that aligns with the denotation of the types being related. Fig. 5 highlights key subtyping rules that do not have analogues in a standard refinement type system. The subtyping rule for symbolic automata (SubAutomata) checks inclusion between the languages of the automata, after performing substitutions consistent with the current typing context. The subtyping rule for (non-intersected) HATs (SubHoare) checks that the inclusion is contravariant over the precondition automata and covariant over the post-condition automata under the same context (i.e., the conjunction of B_1 and B_2 with $A_2; \Box\langle\top\rangle$). The subtyping rules for the intersection of HATs - SubIntLL, SubIntLR, and SubIntR - are standard. The SubIntMerge rule additionally allows the precondition automata of intersected types to be merged. Finally, the subtyping rules for ghost variables either bind a ghost variable in the type context (SubGhostR), or instantiate it to some concrete value (SubGhostL).

A subset of our typing rules[5] is shown in Fig. 6. Note that an effectful computations can only be ascribed a HAT. As mentioned in Example 4.2, our type system is parameterized over Δ, a typing context for built-in operators that provides HATs for both pure (TPOp) and effectful operators (TEOp). This system features the standard subsumption and intersection rules (TSub and TInter), which enables our type system to use fine-grained information about the effect context when typing different control flow paths. The rule TEPur allows a pure term to be treated as a computation that does not perform any effects.

Example 4.6. The add function in Fig. 1 has four possible control flow paths, depending on the effect context under which it is called: (1) the input path exists in the file system (line 6); (2) neither the input path or its parent path exist (line 9); (3) its parent path exists and is a directory (line 13 – 15), or (4) it is not (line 16). The following four automata (recall that the SFA $P_{\texttt{isDir}}$ was defined in Sec. 2) indicate the effect context corresponding to each of these scenarios:

[5]The complete set of typing rules for $\boldsymbol{\lambda}^E$ is included in the supplemental material.

$$A_1 \doteq I_{\mathsf{FS}}(\mathsf{p}) \land P_{\mathsf{exists}}(\mathsf{path})$$
$$A_2 \doteq I_{\mathsf{FS}}(\mathsf{p}) \land \neg P_{\mathsf{exists}}(\mathsf{path}) \land \neg P_{\mathsf{exists}}(\mathsf{parent(path)})$$
$$A_2 \doteq I_{\mathsf{FS}}(\mathsf{p}) \land \neg P_{\mathsf{exists}}(\mathsf{path}) \land P_{\mathsf{exists}}(\mathsf{parent(path)}) \land P_{\mathsf{isDir}}(\mathsf{parent(path)})$$
$$A_3 \doteq I_{\mathsf{FS}}(\mathsf{p}) \land \neg P_{\mathsf{exists}}(\mathsf{path}) \land P_{\mathsf{exists}}(\mathsf{parent(path)}) \land \neg P_{\mathsf{isDir}}(\mathsf{parent(path)})$$

The union of these automata is exactly the representation invariant ($I_{\mathsf{FS}}(\mathsf{p})$), established from the following subtyping relation between their intersection and the return type of τ_{add}:

$$\mathsf{p}{:}\{\nu{:}\mathsf{Path.t} \mid \top\}, \mathsf{path}{:}\{\nu{:}\mathsf{Path.t} \mid \top\}, \mathsf{bytes}{:}\{\nu{:}\mathsf{Bytes.t} \mid \top\} \vdash \bigsqcap_{i=1..4} [A_i]\,\mathsf{bool}\,[I_{\mathsf{FS}}(\mathsf{p})] <: [I_{\mathsf{FS}}(\mathsf{p})]\,\mathsf{bool}\,[I_{\mathsf{FS}}(\mathsf{p})] \quad \text{(SubIntMerge)}$$

Using the TInter and TSub rules, our type system is able to reduce checking add against τ_{add} into checking add against each A_i (i.e., $[A_i]\,\mathsf{bool}\,[I_{\mathsf{FS}}(\mathsf{p})]$). Note that add only adds the given path to the file system in the third case (line 13), whose precondition automata (A_3) indicates that the input path does not exist in the file system ($\neg P_{\mathsf{exists}}(\mathsf{path})$), although its parent path does ($P_{\mathsf{exists}}(\mathsf{parent(path)})$), and is a directory ($P_{\mathsf{isDir}}(\mathsf{parent(path)})$). When coupled with the representation invariant that the parent of a path is always a directory in the file system, our type system is able to ensure that it is safe to perform the **put** operation.

Effectful operator application. TPOpApp is the standard rule for operator application in a typical refinement type system [21]. On the other hand, rule TEOpApp, the rule for effect operator application, with the help of the subsumption rule (TSub), allows operators to have function types whose ghost variables are instantiated properly; moreover, the return type of effect operators is a non-intersected HAT. After ensuring each argument v_i types against the corresponding argument type t_i and substituting all parameters ($\overline{z_i}$) with the supplied arguments ($\overline{v_i}$), the return type of an effectful operator must be in the form $[A]\,t_x\,[A']$ and have exactly the same precondition SFA (A) as the type of the surrounding term. Typing the `let`-body e is similar to TPOpApp, where a new binding $x{:}t_x$ is added to the type context. Moreover, the rule replaces the precondition SFA used to type the body of the `let` with the postcondition SFA from the return type of the effect operator A', so that the body is typed in a context reflecting the use of the effectful operator.

4.3 Type Soundness

The denotation of a SFA $\mathcal{L}(A)$, shown in Fig. 7, is the set of traces accepted or *recognized* by the automata; the denotation of refinement types $[\![t]\!]$ and HATs $[\![\tau]\!]$ are sets of terms; these functions are defined mutually-recursively.

The language of traces ranges over the set of all *well-formed traces* $Tr^{\mathbf{WF}}$, i.e., those only containing events ($\mathbf{op}\,\overline{v_i} = v$) that are well-typed according to the basic type system. As is standard [8], our denotation uses an auxiliary judgement $\alpha, i \models A$ that defines when the suffix of a trace α starting at the i^{th} position is accepted by an automaton. The denotation of an SFA A is the set of (well-formed) traces recognized by A, starting at index 0.

Type denotations. Out type denotations use a basic typing judgement $\emptyset \vdash_{\mathsf{S}} e : s$ that types an expression e according to the standard typing rules of the simply-typed lambda-calculus (STLC), erasing all qualifiers in function argument types. The denotation of pure refinement types is standard [21]. The denotation of ghost variables is similar to function types whose parameters are restricted to base types. The denotation of an intersection type $\tau_1 \sqcap \tau_2$ is the intersection of the denotations τ_1 and τ_2. An expression e belongs to the denotation of a HAT $[A]\,t\,[B]$ iff every trace and value produced by e is consistent with the SFA B and refinement type t, under any effect context accepted by the SFA A. Intuitively, the denotation of a HAT is naturally derived from $\boldsymbol{\lambda}^{E}$'s

Trace Language $\boxed{\alpha, i \models A \quad \mathcal{L}(A) \in \mathcal{P}(\alpha)}$

$$Tr^{\mathbf{WF}} \doteq \{\alpha \mid \forall (\mathbf{op}\,\overline{v_i} = v) \in \alpha. \emptyset \vdash_{\mathsf{S}} \mathbf{op} : \overline{b_i} \to b \land (\forall i. \emptyset \vdash_{\mathsf{S}} v_i : b_i) \land \emptyset \vdash_{\mathsf{S}} v : b\} \qquad \mathcal{L}(A) \doteq \{\alpha \in Tr^{\mathbf{WF}} \mid \alpha, 0 \models A\}$$

$$\alpha, i \models \langle \mathbf{op}\,\overline{x_j} = \nu \mid \phi \rangle \iff \alpha[i] = \mathbf{op}\,\overline{v_j} = v \land \phi\overline{[x_j \mapsto v_j]}[\nu \mapsto v] \qquad \alpha, i \models A \land A' \iff \alpha, i \models A \land \alpha, i \models A'$$

$$\alpha, i \models \langle \phi \rangle \iff \alpha[i] = \mathbf{op}\,\overline{v_j} = v \land \phi \qquad \alpha, i \models A \lor A' \iff \alpha, i \models A \lor \alpha, i \models A'$$

$$\alpha, i \models \bigcirc A \iff \alpha, i{+}1 \models A \qquad \alpha, i \models A_1; A_2 \iff \alpha[i...len(\alpha)] = \alpha_1 {+\!\!+} \alpha_2 \land \alpha_1 \in \mathcal{L}(A_1) \land \alpha_2 \in \mathcal{L}(A_2)$$

$$\alpha, i \models \neg A \iff \alpha, i \not\models A \qquad \alpha, i \models A\,\mathcal{U}\,A' \iff \exists j. i \leq j < len(\alpha). \alpha, j \models A' \land \forall k. i \leq k < j \implies \alpha, k \models A$$

Type Denotation $\boxed{\llbracket t \rrbracket \in \mathcal{P}(e) \quad \llbracket \tau \rrbracket \in \mathcal{P}(e)}$

$$\llbracket \{\nu{:}b \mid \phi\} \rrbracket \doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : b \land \forall \alpha\, \alpha'\, v.\ \alpha \vDash e \xhookrightarrow{\alpha'}{}^{*} v \implies \alpha' = [\,] \land \phi[\nu \mapsto v]\}$$

$$\llbracket x{:}t \to \tau \rrbracket \doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : \lfloor x{:}t \to \tau \rfloor \land \forall v \in \llbracket t \rrbracket.\, e\ v \in \llbracket \tau[x \mapsto v] \rrbracket\}$$

$$\llbracket x{:}b \dashrightarrow \tau \rrbracket \doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : \lfloor \tau \rfloor \land \forall v. \emptyset \vdash_{\mathsf{S}} v : b \implies e \in \llbracket \tau[x \mapsto v] \rrbracket\}$$

$$\llbracket [A]\, t\, [B] \rrbracket \doteq \{e \mid \emptyset \vdash_{\mathsf{S}} e : \lfloor t \rfloor \land \forall \alpha\, \alpha'\, v.\ \alpha \in \mathcal{L}(A) \land \alpha \vDash e \xhookrightarrow{\alpha'}{}^{*} v \implies v \in \llbracket t \rrbracket \land \alpha {+\!\!+} \alpha' \in \mathcal{L}(B)\}$$

$$\llbracket \tau_1 \sqcap \tau_2 \rrbracket \doteq \llbracket \tau_1 \rrbracket \cap \llbracket \tau_2 \rrbracket$$

Type Context Denotation $\boxed{\llbracket \Gamma \rrbracket \in \mathcal{P}(\sigma)}$

$$\llbracket \emptyset \rrbracket \doteq \{\emptyset\} \qquad \llbracket x{:}t, \Gamma \rrbracket \doteq \{\sigma[x \mapsto v] \mid v \in \llbracket t \rrbracket, \sigma \in \llbracket \Gamma[x \mapsto v] \rrbracket\}$$

Fig. 7. Type denotations in $\boldsymbol{\lambda}^E$

operational semantics, as depicted by the following correspondence:

Pure Language: $e \hookrightarrow^{*} v \implies e \in \llbracket \{\nu{:}b \mid \phi\} \rrbracket$ where $\phi[\nu \mapsto v]$ is valid

$\boldsymbol{\lambda}^E$: $\alpha \vDash e \xhookrightarrow{\alpha'}{}^{*} v \implies e \in \llbracket [A]\, \{\nu{:}b \mid \phi\}\, [B] \rrbracket$ where $\alpha \in \mathcal{L}(A), \alpha {+\!\!+} \alpha' \in \mathcal{L}(B)$, and $\phi[\nu \mapsto v]$ is valid

In a pure language with the simple multi-step reduction relation $e \hookrightarrow^{*} v$, refinement types qualify the value v that e produces. In contrast, the multi-step reduction relation of $\boldsymbol{\lambda}^E$ $\alpha \vDash e \xhookrightarrow{\alpha'}{}^{*} v$ depends on the effect context α and emits a trace α' that records the sequence of effects performed by e during evaluation. Thus, HATs use precondition and postcondition automata (i.e., A and B) to qualify the trace before and after a term is evaluated (i.e., α and $\alpha {+\!\!+} \alpha'$).

Type context denotation. The denotation of a type context is a set of *closing substitutions* σ, i.e., a sequence of bindings $\overline{[x_i \mapsto v_i]}$ consistent with the type denotations of the corresponding variables in the type context. The denotation of the empty context is a singleton set containing only the identity substitution $\emptyset$.

Definition 4.7 (Well-formed built-in operator typing context). The built-in operator typing context Δ is well-formed iff the semantics of every built-in operator θ is consistent with its type: $\Delta(\theta) = \overline{x{:}b_x} \dashrightarrow \overline{y{:}t_y} \to \tau \implies \overline{\forall x{:}b_x}.\ \overline{\forall v_y \in \llbracket t_y \rrbracket}.\ (\theta\,\overline{v_y}) \in \llbracket \tau\overline{[y \mapsto v_y]} \rrbracket$.[6]

Theorem 4.8. *[Fundamental Theorem] If the built-in operator type context Δ is well-formed, then for all type contexts Γ, terms v_f and HAT τ, $\Gamma \vdash e : \tau \implies \forall \sigma, \sigma \in \llbracket \Gamma \rrbracket \implies \sigma(e) \in \llbracket \sigma(\tau) \rrbracket$.*[7]

Corollary 4.9. *[Type Soundness] Under a built-in operator type context that is well-formed, if a function f preserves the representation invariant A, i.e. it has the type: $\emptyset \vdash f : \overline{x{:}b_x} \dashrightarrow \overline{y{:}b_y} \to [A]\, t\, [A]$, then for every set of well-typed terms $\emptyset \vdash \overline{v_x : b_x}$ and $\emptyset \vdash \overline{v_y : b_y}$, applying f to $\overline{v_y}$ under an effect context*

[6]Recall that the semantics of an operator is defined by the auxiliary $e \Downarrow v$ and $\alpha \vDash e \Downarrow v$.

[7]A Coq mechanization of this theorem and the type soundness corollary are provided in the supplemental material.

Type Synthesis $\boxed{\Gamma \vdash e \Rightarrow t \quad \Gamma \vdash e \Rightarrow \tau}$ **Type Check** $\boxed{\Gamma \vdash e \Leftarrow t \quad \Gamma \vdash e \Leftarrow \tau}$

$$\frac{\Gamma \vdash^{\textbf{WF}} x{:}b \dashrightarrow \tau \quad \Gamma, x{:}\{v{:}b \mid \top\} \vdash e \Leftarrow \tau}{\Gamma \vdash e \Leftarrow x{:}b \dashrightarrow \tau}\ \textsc{ChkGhost} \qquad \frac{\Gamma \vdash^{\textbf{WF}} \tau_1 \sqcap \tau_2 \quad \Gamma \vdash e \Leftarrow \tau_1 \quad \Gamma \vdash e \Leftarrow \tau_2}{\Gamma \vdash e \Leftarrow \tau_1 \sqcap \tau_2}\ \textsc{ChkInter}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\textbf{WF}} [A_2]\, t_2\, [B_2] \\ \Gamma \vdash e \Rightarrow [A_1]\, t_1\, [B_1] \quad \Gamma \vdash A_2 \subseteq A_1 \\ \Gamma \vdash t_1 <: t_2 \\ \Gamma \vdash (A_2; \Box\langle\top\rangle) \land B_1 \subseteq (A_2; \Box\langle\top\rangle) \land B_2\end{array}}{\Gamma \vdash e \Leftarrow [A_2]\, t_2\, [B_2]}\ \textsc{ChkSub} \qquad \frac{\begin{array}{c}\Gamma \vdash^{\textbf{WF}} [A]\, t\, [B] \quad \Delta(\textbf{op}) = \overline{z_k{:}b_k} \dashrightarrow \overline{y_j{:}t_j} \to \tau_x \\ \forall j. \Gamma \vdash v_j \Leftarrow t_j \\ \Gamma \vdash [A]\ \overline{z_k{:}b_k} \dashrightarrow \overline{y_j{:}t_j} \to \tau_x \leadsto \overline{z_k{:}t_k} \\ \overline{\sqcap [A_i]\, t_i\, [A'_i]} = \tau_x[\overline{y_j \mapsto v_j}] \\ \forall i. \Gamma, \overline{z_k{:}t_k}, x{:}t_i \vdash e \Leftarrow [(A; \Box\langle\top\rangle) \land A'_i]\, t\, [B]\end{array}}{\Gamma \vdash \texttt{let}\ x = \textbf{op}\ \overline{v_i}\ \texttt{in}\ e \Leftarrow [A]\, t\, [B]}\ \textsc{ChkEOpApp}$$

Fig. 8. Selected Bidirectional Typing Rules

α that is consistent with A results in a context α++α' that is also consistent with A:

$$\alpha \vDash (f\ \overline{v_y}) \overset{\alpha'}{\hookrightarrow}{}^{*}\ v \Longrightarrow \alpha +\!\!+ \alpha' \in \mathcal{L}(A[\overline{x \mapsto v_x}][\overline{y \mapsto v_y}]) \land v \in [\![t[\overline{x \mapsto v_x}][\overline{y \mapsto v_y}]]\!]$$

5 TYPING ALGORITHM

Converting our declarative type system into an efficient algorithm requires resolving two key issues. First, typing the use of an effectful operator, e.g., **put** or **exists**, depends on the set of effects in the precondition of the current HAT. The declarative type system can ensure that this automaton has the right shape by applying the TInter and TSub rules at any point in a derivation, but an efficient implementation must perform this conversion more intelligently. Our solution to this problem is to employ a bidirectional type system [11] that closely tracks a context of effects that have preceded the term being typed, and selectively applies the subsumption rule to simplify the context when switching between checking and synthesis modes. Second, we cannot directly use existing SFA inclusion algorithms [7, 12] to implement the check in SubAutomata, because our SFAs may involve non-local variables corresponding to function parameters and ghost variables; these variables can have refinement types. We deal with this wrinkle by integrating the subtyping algorithm for pure refinement types into the existing SFA algorithm.

5.1 Bidirectional Type System

As is standard, our bidirectional type system consists of both type synthesis ($\Rightarrow$) and type checking ($\Leftarrow$) judgments. The bidirectional system features a minor divergence from the declarative rules. While the declarative system was able to use the subsumption rule (TSub) to freely instantiate ghost variables, a typing algorithm requires a more deterministic strategy. Our solution is to treat ghost variables $x{:}b \dashrightarrow$ as instead being qualified by an unknown formula which can be instantiated as needed, based on the information in the current typing context.

With this minor tweak in hand, bidirectionalizing the type system of $\boldsymbol{\lambda}^{E}$ is mostly a matter of choosing appropriate modes for the assumptions of each rule. This is largely straightforward: the type checking rule for ghost variables (ChkGhost) adapts the corresponding declarative rule (TGhost) in the standard way, for example. The rule for checking effectful operations (ChkEOpApp) is similar, although it adopts a fixed strategy for applying the subsumption rule: 1) instead of using TSub to instantiate ghost variables directly, it now relies on an auxiliary function, $\leadsto$, to qualify their values based on the current context, per the discussion above;[8] 2) when the result of the effectful operation **op** is an intersection type ($\overline{\sqcap [A_i]\, t_i\, [A'_i]}$), ChkEOpApp considers each case of the

[8]The details of the instantiation ($\leadsto$) function, and pure refinement subtyping are included in the supplemental material.

intersection, instead of using TSub to focus on a single case; and 3) instead of relying on TSub to align the postcondition automata (A'_i) of **op** with the precondition of the HAT being checked against (A), the rule uses the conjunction of the automata ($(A; \Box\langle\top\rangle) \land A'_i$) as the precondition automata used to check the term that uses the result of **op**.

5.2 SFA Inclusion

Symbolic alphabets allow the character domains of SFAs to be infinite, making them strictly more expressive than standard FAs. This enhanced expressivity comes at a cost, however, as the standard FA inclusion algorithm cannot be directly applied to SFAs. Thankfully, prior work has shown how to reduce an inclusion check over SFAs to an inclusion check over FAs [7, 12]. The high-level approach is to first construct a finite set of equivalence classes over an SFA's infinite character domain, defined in terms of a set of maximal satisfiable Boolean combinations of logic literals (called *minterms*) in the automaton's symbolic alphabet. An alphabet transformation procedure then replaces characters in the original SFA with their corresponding equivalence classes, and replaces the SFA's symbolic alphabet with minterms, thus allowing SFAs to be translated into FAs. As long as the satisfiability of minterms is decidable, so is checking inclusion between two SFAs.

The pre- and post-conditions in HATs have two features that distinguish them from standard SFAs, however: 1) characters are drawn from a set of distinct events and their qualifiers may range over multiple variables, instead of a single local variable; and, 2) event qualifiers may refer to variables included in the typing context Γ, which comes from function parameters and ghost variables. Our language inclusion algorithm accounts for all these differences by extending the existing algorithm outlined above to incorporate a subtyping check between pure refinement types.

Candidate Minterms. The first step in checking SFA inclusion is to construct the equivalence classes used to finitize its character set. With HATs, these characters range over events triggered by the use of effectful operations $\textbf{op}\,\overline{v} = v$, and return events $\textbf{ret}\,v$. As each class of events is already disjoint (e.g., **get** and **put** events will never overlap), we only need to build minterms that partition individual classes. Then the minterms of events, which are in the form $\langle \textbf{op}\,\overline{x_i} = \nu \mid (\bigwedge \overline{l}) \land (\bigwedge \overline{\neg l})\rangle$, are constructed in the standard manner from atomic predicates: for $\langle \textbf{op}\,\overline{x_i} = \nu \mid \phi\rangle$, we collect all literals $\overline{l}$ used to qualify **op**, and then construct a maximal set of satisfiable Boolean combinations of $\overline{l}$; on the other hand, the literals of atomic predicate $\langle\phi\rangle$ are shared by all effect operators. Notably, these literals may contain variables bound in the current typing context.

Satisfiability Check. Importantly, the soundness of the alphabet transfer procedure requires that *only* satisfiable minterms are used, so that every transition in the output FA corresponds to an actual transition in the original SFA. To understand how this is done, observe that the typing context plays a similar role in checking both the satisfiability of minterms and the subtyping relation. Thus, the algorithm reduces checking the satisfiability of $\langle \textbf{op}\,\overline{x_i} = \nu \mid (\bigwedge \overline{l}) \land (\bigwedge \overline{\neg l})\rangle$ to checking whether the refinement type $\{\nu{:}b \mid (\bigwedge \overline{l}) \land (\bigwedge \overline{\neg l})\}$ is *not* a subtype of $\{\nu{:}b \mid \bot\}$, that is $\Gamma, \overline{x_i{:}b_i} \nvdash \{\nu{:}b \mid (\bigwedge \overline{l}) \land (\bigwedge \overline{\neg l})\} <: \{\nu{:}b \mid \bot\}$. Since $\{\nu{:}b \mid \bot\}$ is uninhabited, this subtype check fails precisely when $(\bigwedge \overline{l}) \land (\bigwedge \overline{\neg l})$ is satisfiable under Γ.

Inclusion Check. Equipped with the set of satisfiable minterms, the inclusion check is a straightforward application of the textbook inclusion check between the FAs resulting from the standard alphabet transfer algorithm.

Theorem 5.1. *[Soundness of Algorithmic Typing] Given a well-formed built-in typing context Δ, type context Γ, term e, and HAT τ, $\Gamma \vdash e \Leftarrow \tau \implies \Gamma \vdash e : \tau$.*[9]

[9]The details of SFA inclusion algorithm and the proof of Theorem 5.1 can be found in the supplementary material.

Table 1. Experimental results of using Marple to verify representation invariants of ADTs. Benchmarks from prior works are annotated with their source: Okasaki [32](†), Hanoi [27](*). The backing effectful libraries are drawn from a verified OCaml library [5].

ADT	Library	#Method	#Ghost	s_I	t_{total} (s)	#Branch	#App	#SAT	$\#FA_\subseteq$	avg. s_{FA}	t_{SAT} (s)	$t_{FA_\subseteq}$ (s)
Stack†	LinkedList	7	0	4	5.94	4	7	297	7	98	1.63	0.06
	KVStore	7	1	9	17.88	4	10	874	17	226	5.78	0.23
Queue†	LinkedList	6	0	4	4.88	4	9	190	4	96	1.08	0.06
	Graph	6	1	24	28.24	5	12	1212	14	525	8.19	0.86
Set*	Tree	5	0	12	27.12	5	12	1589	11	531	9.34	0.53
	KVStore	3	1	9	2.39	3	5	245	6	160	1.51	0.06
Heap*	Tree	7	0	12	25.71	5	12	1589	11	531	9.33	0.52
	LinkedList	6	0	4	8.84	4	8	497	8	118	3.03	0.08
MinSet	Set	4	1	28	10.55	3	6	612	17	294	4.19	1.58
	KVStore	4	1	25	32.84	5	8	2227	23	519	10.93	9.34
LazySet	Tree	6	0	12	27.10	5	12	1589	11	531	9.33	0.57
	Set	4	1	9	1.42	2	3	101	4	106	0.57	0.04
	KVStore	5	1	9	3.10	3	5	245	6	160	1.49	0.06
FileSystem	Tree	6	1	20	58.80	3	8	2085	17	652	14.15	2.21
	KVStore	4	1	17	157.27	4	10	8144	43	481	56.64	16.54
DFA	KVStore	5	2	18	42.62	3	3	3604	25	228	18.84	1.60
	Graph	5	2	11	78.44	4	4	3625	27	225	25.53	3.42
ConnectedGraph	Set	5	2	9	80.46	4	4	3889	50	357	27.39	12.80
	Graph	5	1	20	176.89	4	3	1349	17	360	19.19	53.90

6 IMPLEMENTATION AND EVALUATION

We have implemented a tool based on the above approach, called Marple, that targets ADTs implemented in terms of other effectful libraries. Marple consists of approximately 12K lines of OCaml and uses Z3 [9] as its backend solver for both SMT and FA inclusion queries.[10]

Marple takes as an input the implementation of an ADT in OCaml, enhanced signatures of ADT operations that include representation invariants expressed as HATs, and specifications of the supporting libraries as HATs (e.g., the signatures in Example 4.2). The typing context used by Marple includes signatures for a number of (pure) OCaml primitives, including the pure operators listed in Fig. 2. Marple also includes a set of predefined method predicates (i.e., *mp* in Fig. 4) that allow qualifiers to capture non-trivial datatype properties. For example, the method predicates `isDir(x)` and `isDel(y)` from the motivating example in Sec. 2 encode that `x` holds the contents of a directory, and that `y` is marked as deleted, respectively. The semantics of method predicates are defined via a set of lemmas in FOL, in order to enable automated verification; e.g., the axiom $\forall$`x.isDir(x)` $\implies \neg$`isDel(x)` encodes that a set of bytes cannot simultaneously be an active directory and marked as deleted.

We have evaluated Marple on a corpus of complex and realistic ADTs drawn from a variety of sources (shown in the title of Table 1), including general datatypes such as stacks, queues, sets, heaps, graphs, as well as custom data structures including the Unix-style file system from Sec. 1 and the deterministic finite automata described in Sec. 4. Table 1 presents the results of this evaluation. Each datatype includes multiple implementations using different underlying libraries (Library column), each of which provides a distinct set of operations. Our backing APIs include linked lists, a persistent key-value store, sets, trees, and graphs, each of which induces a different encoding of the target ADT's representation invariant.

Table 1 divides the results of our experiments into two categories, separated by the double bars; the first gives comprehensive results about analyzing the entire datatype, while the second reports

[10]Our supplemental material provides a docker image that contains the source code of Marple and all our benchmarks.

information about the most complex method of each ADT implementation.[11] The first group of columns describes broad characteristics of each ADT, including its number of methods (#Method), the number of ghost variables in its representation invariant (#Ghost), and the size of the formula encoding the representation invariant (s_I), after it is desugared into a symbolic regular expression. The next column reports the total time needed to verify all the method of each ADT (t_{total}). Marple performs quite reasonably on all of our benchmarks, taking between 1.42 to 176.89 seconds for each ADT. As expected, more complicated ADT implementations (i.e., those with larger values in the #Method, #Ghost, and s_I columns), take longer to verify.

The first group of columns in the second half of the table list features relevant to the complexity of the most demanding method in the ADT. These methods feature between 3 and 5 branches (#Branch) and 3 to 12 uses of built-in operators (#App). The last two groups of columns present type checking results for these methods. The #SAT and $\#FA_{\subseteq}$ columns list the number of SMT queries and finite automata (FA) inclusion checks needed to type check the method. The next column (avg. s_{FA}) gives the average number of transitions in the finite state automata (FA) after the alphabet transformation described in Sec. 5. These numbers are roughly proportional to code complexity (column #Branch and #App) and invariant complexity (column #Ghost and s_I) — intuitively, programs that have more uses of operators and larger specifications, require more queries and produce more complicated FAs. The last two columns report verification time for the method. These times are typically dominated by the SFA inclusion checks (ranging from .63 to 73.09 seconds) that result from minterm satisfiability checks during the alphabet transformation phase (t_{SAT}) and FA inclusion checks ($t_{FA_{\subseteq}}$). Unsurprisingly, generating more queries (both #SMT and $\#FA_{\subseteq}$) result in more time spent checking minterm satisfiability and FA inclusion. Our results also indicate that FA inclusion checks are not particularly sensitive to the size of the FA involved; rather, the cost of these checks corresponds more closely to the deep semantics of the representation invariant, in which the choice of the underlying library is an important factor. Taking the FileSystem benchmark as an example, both Tree and KVStore implementations lead to similar sizes for the invariant, in term of #Ghost. s_I, and FA (avg.s_{FA}). However, the use of the former library results in a significantly shorter verification time since the relevant property captured by the invariant "*every child has its parent in the file system*" is naturally supported by the Tree library; the only remaining verification task in this case for the client to handle involves ensuring these parents are indeed directories in the file system. In contrast, the KVStore provides no such guarantees on how its elements are related, requiring substantially more verification effort to ensure the invariant is preserved.

7 RELATED WORK

Representation Invariant Inference. [27] develops a data-driven CEGIS-based inference approach, while [26] develop a solution derived from examining testcases. Solvers used to infer predicates that satisfy a set of Constrained Horn Clauses (CHCs) [13, 19, 44] can also be thought of as broadly addressing similar goals. Our focus in this paper is orthogonal to these efforts, however, as it is centered on the automated verification of user-specified representation invariants. In our setting, these invariants are expressed using SFAs and are checked against functional programs that nonetheless interact with stateful libraries that manage opaque state. We are unaware of any prior work that has directly addressed solutions to this important class of safety problems.

Effect Verification. There has been significant prior work that considers the type-based verification of functional programs extended with support for (structured) effectful computation. Ynot [29] and F* [38] are two well-known examples whose type specifications allow fine-grained effect tracking of stateful functional programs. These systems allow writing specifications that leverage type

[11] Our supplemental material includes a full table listing the corresponding information for each ADT method.

abstractions like the Hoare monad [30] or the Dijkstra monad [2, 25, 39]. For example, specifications using the Hoare monad involve writing type-level assertions in the form of pre- and post-conditions over the heaps that exist before and after an effectful computation executes. The Dijkstra monad generalizes this idea by allowing specifications to be written in the style of a (weakest precondition) predicate transformer semantics, thus making it more amenable for improved automation and VC generation. While our goals are broadly similar to these other efforts, our setup (and consequently, our approach) is fundamentally different. Because we assume that the implementation of effectful operations are hidden behind an opaque interface, and that the specifications of these operations are not tailored for any specific client application, our verification methodology must necessarily reason about effectful library actions, indirectly in terms of the *history* of calls (and returns) to library methods made by the client, rather than in terms of predicates over the concrete representation of the state. Our two key observations are that (a) verification in this setting benefits from specifying pre- and post-conditions on an effectful computation in terms of such histories, and (b) that SFAs (and their type embodiment as HATs) are well-suited to serve as their underlying representation. One important way that our significantly different representation choice impacts our verification methodology is that unlike e.g., the Dijkstra monad that uses a predicate transformer semantics to establish a relation between pre- and post-states of an effectful action, our typing of interaction history in terms of HATs allows us to simply reuse a standard subtying relation (suitably adapted), instead. Consequently, the implementation of our typing algorithm is amenable to a significant degree of automation, on par with other refinement type systems like Liquid Haskell [41]. Less aligned with our approach are efforts to equip low-level imperative programs with refinement types [24, 33, 34]. These approaches need to deal with the pervasive heap changes that occur in a programming model that does not cleanly delineate pure and effectful computations, and must therefore incorporate notions of separation, ownership, and strong updates directly into the type system. While these features are highly relevant to safety verification in their setting, our focus is on the verification of client-specified representation invariants that interact with effectful libraries in which state is hidden behind library method interfaces.

Type and Effect Systems. Type and effect systems comprise a broad range of techniques that provide state guarantees about both *what* values a program computes and *how* it computes them. These systems have been developed for a number of different applications: static exception checking in Java [20], ensuring effects are safely handled in the languages with algebraic effects [3], and reasoning about memory accesses [16], in order to ensure, e.g., programs can be safely parallelized [1]. Notably, a majority of these systems are agnostic to the order in which these effects are produced: the effect system in Java, for example, only tracks whether or not a computation may raise an exception at run-time.

In contrast, *sequential* effect systems [40] provide finer grained information about the order in which a computation's effects may occur. More closely related to this work are type and effect systems that target *temporal* properties on the sequences of effects a program may *produce*. For example, Skalka and Smith[36] present a type and effect system for reasoning about the shape of histories (i.e., finite traces) of events embedded in a program, with a focus on security-related properties expressed as regular expressions augmented with fixed points. Their specification language is not rich enough to capture data-dependent properties: the histories of a conditional expression must include the histories of both branches, even when its condition is a variable that is provably false. Koskinen and Terauchi[23] present a type and effect system that additionally supports verification properties of infinite traces, specified as Büchi automata. Their system features refinement types and targets a language similar to λ^E. Their effect specifications also include a second component describing the sets of *infinite* traces a (non-terminating) program may produce, enabling it to reason about liveness properties. Unlike λ^E, however, events in their system are not allowed to refer to program variables. This restriction was later lifted by Nanjo *et. al*[31], whose results support value-dependent predicates in effect specifications written in a first-order fixpoint logic. To solve the resulting verification conditions, they introduce a novel proof system for this logic. More recently, [35] consider how to support richer control flow structures, e.g., delimited continuations, in such an effect system. Notably, to the best of our knowledge, all of these effect systems lack an analogue to the precondition automaton of HATs, and instead have to rely on the arguments of a dependently typed function to constrain its set of output traces, as opposed to constructing and managing the context of previously seen events and their results.

While our focus has been on the development of a practical type system for the purposes of verification, there have also been several works that provide foundational characterizations of sequential effect systems. Most of these works adopt a semantics-oriented approach and seek to develop categorical semantics for effectful languages [22, 28, 40]. More closely related is the recent work of Gordon[17], which defines a generic model of sequential effects systems in terms of a parameterized system. The key idea is to represent effects as a *quantale*, a join semi-lattice equipped with a top element and whose underlying set is also a monoid. The proposed type system is parameterized over an arbitrary effect quantale; this system is proven to be safe for any appropriate instantiation of this parameter. Importantly, this framework includes a context of previously seen events, similar to the precondition of a HAT. The paper describes several concrete quantales that enable prior works to be embedded in this system (e.g., [14]).

Type-Based Protocol Enforcement. Because HATs express a history of the interaction between a client and an effectful library, they naturally serve as a form of protocol specification on client behavior, although the kinds of protocols we consider in this work are limited to those that are relevant to defining a representation invariant, i.e., protocols which capture constraints on library methods sufficient to ensure a desired invariant of a client. In this sense, HATs play a similar role as typestate in object-oriented languages [4, 10, 37], which augments the interface of objects with coarse-grained information about the state of an object, in order to constrain its set of available methods. Typestate specifications differ in obvious ways from HATs, most notably with respect to the level of granularity that is expressible; in particular, HATs are designed to capture fine-grained interactions between libraries and clients that are necessary to specify useful representation invariants.

8 CONCLUSIONS

This paper explores the integration of SFAs within a refinement type system as a means to specify and verify client-specific representation invariants of functional programs that interact with effectful libraries. Our key innovation is the manifestation of SFAs as HATs in the type system, which allows the specification of context traces (preconditions) as well as computation traces (postconditions) that allow us to characterize the space of allowed client/library interactions and thus enable us to prove that these interactions do not violate provided representation invariants. HATs integrate cleanly within a refinement type system, enjoy pleasant decidablity properties, and are amenable to a high-degree of automation. Extending our current formalization to support richer automata classes (e.g., symbolic visibly pushdown [6] or Büchi automata) to support a larger class of safety properties or control abstractions (e.g., algebraic effects [3]) for which even more sophisticated specifications than can be expressed using HATs are required is a subject for future work.

REFERENCES

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. 2011. Semantics of Transactional Memory and Automatic Mutual Exclusion. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 2 (jan 2011), 50 pages. https://doi.org/10.1145/1889997.1889999

[2] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 515–529. https://doi.org/10.1145/3009837.3009878

[3] Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*, Reiko Heckel and Stefan Milius (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[4] Eric Bodden and Laurie Hendren. 2012. The Clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer* 14 (2012), 307–326.

[5] Arthur Charguéraud, Jean-Christophe Filliâtre, Mário Pereira, and François Pottier. 2017. VOCAL – A Verified OCaml Library. ML Family Workshop.

[6] Loris D'Antoni and Rajeev Alur. 2014. Symbolic Visibly Pushdown Automata. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 209–225.

[7] Loris D'Antoni and Margus Veanes. 2014. Minimization of Symbolic Automata. *SIGPLAN Not.* 49, 1 (jan 2014), 541–553. https://doi.org/10.1145/2578855.2535849

[8] Giuseppe De Giacomo and Moshe Y. Vardi. 2013. Linear Temporal Logic and Linear Dynamic Logic on Finite Traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (Beijing, China) *(IJCAI '13)*. AAAI Press, 854–860.

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[10] Robert DeLine and Manuel Fähndrich. 2004. Typestates for Objects. In *ECOOP 2004–Object-Oriented Programming: 18th European Conference, Oslo, Norway, June 14-18, 2004. Proceedings 18*. Springer, 465–490. https://doi.org/10.1007/978-3-540-24851-4_21

[11] Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952

[12] Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*. Springer, 47–67.

[13] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE learning for synthesizing invariants and contracts. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 131:1–131:25. https://doi.org/10.1145/3276501

[14] Cormac Flanagan and Shaz Qadeer. 2003. A Type and Effect System for Atomicity. *SIGPLAN Not.* 38, 5 (may 2003), 338–349. https://doi.org/10.1145/780822.781169

[15] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

[16] David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/319838.319848

[17] Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 4 (apr 2021), 79 pages. https://doi.org/10.1145/3450272

[18] John Hatcliff and Olivier Danvy. 1994. A Generic Account of Continuation-Passing Styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 458–471. https://doi.org/10.1145/174675.178053

[19] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj S. Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–7. https://doi.org/10.23919/FMCAD.2018.8603013

[20] Java 2013. Java Platform Standard Edition 7 Documentation. https://docs.oracle.com/javase/7/docs/

[21] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Found. Trends Program. Lang.* 6, 3-4 (2021), 159–317. https://doi.org/10.1561/2500000032

[22] Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*.

Association for Computing Machinery, New York, NY, USA, 633–645. https://doi.org/10.1145/2535838.2535846

[23] Eric Koskinen and Tachio Terauchi. 2014. Local Temporal Reasoning. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)* (Vienna, Austria) *(CSL-LICS '14)*. Association for Computing Machinery, New York, NY, USA, Article 59, 10 pages. https://doi.org/10.1145/2603088.2603138

[24] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. 7, PLDI (2023). https://doi.org/10.1145/3591283

[25] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (jul 2019), 29 pages. https://doi.org/10.1145/3341708

[26] Muhammad Zubair Malik, Aman Pervaiz, and Sarfraz Khurshid. 2007. Generating Representation Invariants of Structurally Complex Data. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 34–49. https://doi.org/10.1007/978-3-540-71209-1_5

[27] Anders Miltner, Saswat Padhi, Todd D. Millstein, and David Walker. 2020. Data-driven Inference of Representation Invariants. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1–15. https://doi.org/10.1145/3385412.3385967

[28] Alan Mycroft, Dominic Orchard, and Tomas Petricek. 2015. Effect Systems Revisited–Control-Flow Algebra and Semantics. In *Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays on Semantics, Logics, and Calculi - Volume 9560*. Springer-Verlag, Berlin, Heidelberg, 1–32. https://doi.org/10.1007/978-3-319-27810-0_1

[29] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. 2008. Ynot: Dependent Types for Imperative Programs. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/1411204.1411237

[30] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare Type Theory, Polymorphism and Separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911. https://doi.org/10.1017/S0956796808006953

[31] 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* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 759–768. https://doi.org/10.1145/3209108.3209204

[32] Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press, USA.

[33] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-Level Liquid Types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 131–144. https://doi.org/10.1145/1706299.1706316

[34] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 158–174. https://doi.org/10.1145/3453483.3454036

[35] 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, Article 71 (jan 2023), 32 pages. https://doi.org/10.1145/3571264

[36] Christian Skalka and Scott Smith. 2004. History Effects and Verification. In *Programming Languages and Systems*, Wei-Ngan Chin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–128.

[37] Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-Class State Change in Plaid. *SIGPLAN Not.* 46, 10 (oct 2011), 713–732. https://doi.org/10.1145/2076021.2048122

[38] Nikhil Swamy, Guido Martinez, and Aseem Rastogi. 2023. *Proof-Oriented Programming in F**. https://fstar-lang.org/tutorial/proof-oriented-programming-in-fstar.pdf

[39] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 387–398. https://doi.org/10.1145/2491956.2491978

[40] Ross Tate. 2013. The Sequential Semantics of Producer Effect Systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2429069.2429074

[41] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. *SIGPLAN Not.* 49, 9 (aug 2014), 269–282. https://doi.org/10.1145/2692915.2628161

[42] Margus Veanes. 2013. Applications of Symbolic Finite Automata. In *Implementation and Application of Automata*, Stavros Konstantinidis (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–23.

[43] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. 2010. Symbolic Automata Constraint Solving. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Andrei Fermüller, Christian G.and Voronkov (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 640–654.

[44] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 707–721. https://doi.org/10.1145/3192366.3192416