Modular Effects in Haskell through Effect Polymorphism and Explicit Dictionary Applications

A New Approach and the µVeriFast Verifier as a Case Study

Dominique Devriese
Software Languages Lab - Vrije Universiteit Brussel - Belgium
dominique.devriese@vub.be

Abstract
In applications with a complex structure of side effects, effects should be dealt with modularly: components should be programmed against abstract effect interfaces that other components can instantiate as required, and reusable effect patterns should be factored out from the rest of the application. In this paper, we study a new, general approach to achieve this in Haskell by combining effect polymorphism and the recently proposed coherent explicit dictionary applications. We demonstrate the elegance and generality of our approach in µVeriFast: a Haskell-based reimplementation of the semi-automatic separation-logic-based verification tool VeriFast. This implementation features a complex interplay of advanced side effects: a backtracking search of program paths with angelic and demonic non-determinism, interaction with an underlying off-the-shelf SMT solver, and mutable state that is either backtracked or not during the search. Our use of effect polymorphism improves over the current non-modular implementation of VeriFast, allows us to nicely factor out the backtracking search pattern as a new Assume/Assert monad, and enables advanced features involving effects, such as the non-intrusive addition of a graphical symbolic debugger based on delimited continuations.

CCS Concepts • Theory of computation → Control primitives; • Software and its engineering → Functional languages: Polymorphism.

Keywords modular effects, Haskell, effect polymorphism, monads, separation logic, symbolic execution, backtracking

ACM Reference Format:

1 Introduction
In this paper, we propose and study an approach to deal modularly with side effects in Haskell. To motivate and explain it, we use VeriFast [15] as a case study. This is a semi-automatic separation-logic-based verification tool for C and Java programs. Figure 1 shows a C function with pre- and postconditions in VeriFast syntax. The function takes an integer pointer, increments its value and returns the old value. The pre- and postconditions of f require exclusive ownership of the memory location, define its value before and after execution and specify the function result.

To verify components like f, VeriFast uses an approach based on symbolic execution with an underlying SMT solver. All execution paths of a program are searched in a backtracking search. During this search, VeriFast keeps track of the symbolic heap and the path condition. The symbolic heap is the list of atomic separation logic assertions that are available for use at the current execution point. When verifying a function, the symbolic heap is initially filled with the assertions in the precondition and is updated whenever a statement is symbolically executed. For example in Figure 1, the symbolic heap would contain just the predicate \( x \mapsto n \) before execution of line 5. After line 5, it would additionally contain \( y \mapsto 5 \) etc. The path condition is a logical assertion that tracks purely logical information learned from the execution path being explored. For example in Figure 1, the path condition will contain \( n \neq 15 \) in the then branch and \( n = 15 \) in the else branch.

Throughout verification, VeriFast uses an SMT solver as an oracle for logical queries. This solver is incrementally fed the path condition as an assumption and continuously asked to verify logical assertions. For example, on line 9 of Figure 1, after the assignment, we have symbolic heap \( x \mapsto 16 \) and path condition \( n = 15 \). Because the program returns, VeriFast will assert that result = 15 and try to satisfy the postcondition. Concretely, it will use \( x \mapsto 16 \) to satisfy \( x \mapsto n + 1 \) and verify that result = n. The solver will confirm result = n and 16 = n + 1 using the assumption n = 15.

\( ^{1} \)VeriFast uses either Z3 [10] or a simpler, custom-built solver called Redux.
Figure 1. An example program that can be verified using VeriFast.

```haskell
int f(int * x) {
    //@ requires x \mapsto \mathtt{?n}
    //@ ensures x \mapsto n+1 &*\& result = n
    if( *x != 15 ) {
        int *y = malloc(\*sizeof(int)); *y = 5;
        free(y); return (*x++);
    } else {
        *x = 16; return 15;
    }
}
```

Most non-deterministic choices during VeriFast’s back-tracking search are demonic: verification must separately succeed for both choices. For example, in Figure 1, VeriFast will verify the if-statement on lines 5-10 by non-deterministically choosing the then or else branch. After verifying a branch, it will backtrack and explore the other, and report success only if both branches verify. However, when VeriFast has multiple ways to satisfy an assertion, this choice is made angelically: VeriFast will simply try all choices until one succeeds and then continue verification. This choice is angelic: it suffices that one choice succeeds and no others will be tried. Angelic and demonic non-determinism have been used to formalise VeriFast’s operation [16, 35].

VeriFast’s implementation in OCaml has to deal with the complex interplay of side effects used:

- the backtracking search across angelic and demonic branches
- the state of the symbolic heap, the current mapping from program variables to their logical interpretation
- an environment of previously processed function declarations and their contracts
- the state of the underlying SMT solver
- C control effects like `return` and `break` statements
- logging and internal statistics gathering

These different effects interact in non-trivial ways. For example, mutable state and the underlying SMT solver need to be rolled back when backtracking over a branch, but logs and statistics should not be.

To manage these, VeriFast is written in a manual state-passing, continuation-passing style. Figure 2 shows a simplified excerpt of the actual VeriFast codebase and shows how an `if` statement is verified. We recommend that you do not try to understand the code snippet in detail, but simply notice the points listed below. First, `verify_stmt` receives quite a few arguments:

- three continuations: `lblenv` defines what to do for a jump to a label, `tcont` represents the regular continuation and `return_cont` defines what to do when a return statement is reached (typically: skip subsequent statements and verify the function’s postcondition).
- an environment variable (`funcmap`) that provides information about declared functions and their contracts.
- three mutable variables (`tenv, h` and `env`) that define the mappings of variables to their type and their interpretation as logical term, and the current symbolic heap. Note on line 5, how the continuation `tcont` is passed an updated version of `env` (removing variables added in the branches). Note also how the two invocations of `verify_block` receive the same value for `env`, so that, when the first finishes, the environment will effectively be rolled back before executing the second.
- the statement `s` to be verified

The manual state- and continuation-passing in Figure 2 generates complexity that is unrelated to the task at hand (verifying an `if`-statement), tedious and error-prone. Additionally, it couples the code to a fixed set of side effects so that, for example, adding extra backtrackable state requires refactoring large parts of the codebase. The goal of this paper is to improve such code by treating effects more modularly. Specifically, it should be oblivious to effects that it is not itself concerned with, like state and continuations. Additionally, other code should be free to instantiate the effects in different ways, for example, adding extra backtrackable state or delimited continuations (see Section 4.5).

In this paper, we make the following contributions:

- `\mu`VeriFast the Problem: a description of the complex combination of effects with non-trivial interplays in a real-life application. `\mu`VeriFast forms a challenging benchmark for modular effect frameworks.
- modular effects through effect polymorphism and explicit dictionary applications: a new, general approach to achieve modular effects in Haskell by combining the existing approach of effect-polymorphism with explicit dictionary applications [38].
- `\mu`VeriFast the Solution: a proof-of-concept implementation of VeriFast that solves `\mu`VeriFast the Problem using the proposed approach for modular effects. We demonstrate how it enables new advanced uses of effects like the non-intrusive addition of a graphical debugger based on delimited continuations.
- some secondary contributions like the AssumeAssert monad, which elegantly factors out VeriFast’s backtracking search and turns it into a reusable effect pattern. It offers angelic and demonic non-determinism, and can be combined with arbitrary underlying backtrackable effects (specifically mutable state and the Z3 SMT solver).

For presentation, the code from `\mu`VeriFast shown in this paper simplifies the real implementation, included as supplementary material.

---

2Available as open source: https://github.com/verifast/verifast.
3We have already removed ten further arguments for simplicity.
2 Modular Effects using Effect Polymorphism

In this paper, we will achieve modular effects using effect polymorphism, a widely-used technique that is known under a variety of names in the Haskell and Scala communities: ML-style effects [see, e.g., 9] (after the use of type classes from the Monad Transformer Library by Liang et al. [24]), Tagless final [see, e.g., 2, 34] (after Carette et al. [8], who used a related approach for representing lambda calculi) and the van Laarhoven free monad [see, e.g., 1, 27] (because it resembles an encoding of lenses by van Laarhoven [33]).

Consider programs interacting with mutable state through the MonadState type class:

```haskell
class Monad m => MonadState s m where
  get :: m s
  put :: s -> m ()

doubleState :: MonadState Int m => m ()
doubleState = do x <- get
                 put (x + x)
```

This MonadState m plays the role of an abstract effect interface that offers effect primitives get and put to read and write a mutable variable of type s. These primitives produce side-effecting computations in m that can be combined with other computations using the Monad primitives return and bind (>>=) and the do-notation. The program doubleState is not defined in Haskell’s IO monad or any other particular monad. Instead, it is effect-polymorphic: it is defined to work in an arbitrary monad m on the condition that there is an instance of the type class MonadState Int for m. This means that other code can invoke doubleState in any monad of its choosing and provide its own implementation of the effect primitives in MonadState Int for that monad.

2.1 The Lack of Local Instances

Unfortunately, while effect polymorphism offers modular effects, it is not fully general and expressive in Haskell. Imagine that we want to invoke doubleState in Haskell’s IO monad. The function’s MonadState Int m constraint really only says that the function needs access to a mutable state variable of type Int. We would like to use the IO monad’s ML-style mutable references IORef Int to instantiate this variable:

```haskell
client :: IO ()
client = do r <- newIORef 0
          let put_ x = writeIORef r x
              get_ = readIORef r

? -- invoke doubleState with put_ and get_?
```

In the above function client, we allocate a fresh mutable reference r with initial value 0 and use it to implement functions put_ and get_ that we want to instantiate MonadState Int with. Unfortunately, this is not possible in Haskell. The problem is that the mutable reference r exists only locally, inside the monadic computation. However, in Haskell, an instance must necessarily be top-level, and as such cannot refer to local variables like r. As a result, the MonadState type class is only ever used with other monads like the State monad (State s a = s -> (a, s)). It cannot be used with the IO monad, and its physical-memory-backed IORefs.

The restriction to top-level, closed instances is general, but can be worked around in four ways, and we will discuss two in detail. The others (reflecting values into types [22, 30] and the ReaderT pattern [31]) are discussed in Section 6.

2.2 Explicit Effect Dictionaries

The first solution is to avoid type classes. At the cost of some extra verbosity, we can replace the class MonadState Int m with a data type StateD Int m:

```haskell
data StateD s m = StateD { getM :: m s
                            , putM :: s -> m () }

doubleState :: Monad m => StateD Int m => m ()
doubleState sd = do x <- getM sd
                    putM sd (x + x)
```

In other words, by not relying on type class resolution to pass around effect dictionaries in the background, but doing it manually instead, we can side-step the restriction above:

```haskell
client :: IO ()
client = do r <- newIORef 0
            let put_ x = writeIORef r x
                get_ = readIORef r
            doubleState (StateD get_ put_)
```
This approach is a general solution to achieve modular effects and standard Haskell. However, manually passing around dictionaries in real code can be verbose and tedious.

Note, by the way, that a $\text{StateD}$ a m value is similar to a mutable reference of type a in monad m, but more flexible. For example, we can use a $\text{Lens}$ a b (defining how to inspect and modify a b value inside a values [12]), to convert a $\text{StateD}$ a m into a $\text{StateD}$ b m.

\[
\text{lensStateD :: Monad m => Lens a b \rightarrow StateD a m \rightarrow StateD b m}
\]

### 2.3 Implicit Effect Dictionaries

We contribute a different way to remove the restriction of effect polymorphism in Haskell, based on the GHC extension DictionaryApplications, recently proposed by Winant and Devriese [38]. This extension removes the restriction of Haskell type classes mentioned above and allows one to instantiate a type class constraint like $\text{MonadState Int}$ m with an explicit dictionary. It imposes certain conditions on explicit dictionary instantiations to preserve desirable Haskell properties like global instance uniqueness and coherence.

Using this extension, no modifications to $\text{doubleState}$ are needed to invoke it with a $\text{IORef}$. Instead, we can satisfy the $\text{MonadState}$ constraint with an explicit dictionary of type $\text{MonadState Dict Int}$ m. The current implementation uses double parentheses as temporary syntax to denote such an application (when the extension is enabled). \(^4\)

\[
\text{getMonadD :: } \forall m. \text{Monad m \Rightarrow Monad.D dict m}
\]

\[
\text{client :: IO ()}
\text{client = do r \leftarrow newIORef 0}
\text{let put_ x = writeIORef r x}
\text{get_ = readIORef r}
\text{doubleState ((MonadState Dict}
\text{getMonadD get_ put_.)})
\]

The $\text{MonadState}$ dictionary is constructed from an implementation for get and put, as well as a dictionary for the parent $\text{Monad}$ constraint. This parent dictionary is obtained from regular constraint resolution using a function $\text{getMonadD}$.

In this paper, we consistently use effect polymorphism for dealing modularly with effects. For most effect interfaces, we use type classes like $\text{MonadState}$ and explicit dictionary applications. This allows us to keep abstract effect dictionaries around as type class constraints when we just want to pass them around regularly (like in $\text{doubleState}$), but turn them into explicit dictionaries that can be manipulated when we want to play more complicated tricks (see below).

In some situations, the more verbose explicit dictionaries of the previous section 2.2 are in fact preferable. We do this, for example, for state dictionaries, because we can then easily use multiple state variables of the same type at the same time. In other situations, we avoid implicit effect dictionaries because explicit dictionary applications would not be accepted for them, often due to limitations of GHC or the DictionaryApplications extension (see Section 5.2).

### 2.4 Allocating Fresh Effects

To actually implement an effect, we often steer clear of classic Haskell monads like $\text{State}$ or $\text{Writer}$. Instead, the state or output effects which those monads implement, can be obtained more efficiently\(^5\) and more flexibly on top of physical hardware memory, as offered by $\text{IO}$ or $\text{ST}$.

As we have seen above, Haskell’s standard $\text{IO}$ monad allows dynamically allocating fresh mutable variables as $\text{IORef}$s. To model this dynamic allocation of fresh mutable state variables, we use the following $\text{AllocD}$ effect interface (an allocator) or its type class variant $\text{MonadAlloc}$:

\[
\text{data AllocD m = AllocD { alloc :: } \forall s. s \rightarrow m (StateD s m)}
\]

\[
\text{class Monad m \Rightarrow MonadAlloc m where}
\text{alloc :: } \forall s. s \rightarrow m (StateD s m)
\]

The method $\text{alloc}$ in interface $\text{AllocD}$ m allocates a fresh mutable variable with a given initial value and returns a $\text{StateD}$ interface for manipulating it. Explicit dictionary applications are currently not allowed for $\text{MonadAlloc}$ (see Section 5.2), so we use $\text{AllocD}$ instead.

$\text{AllocD}$ can be implemented for $\text{IO}$ using $\text{newIORef}$:

\[
\text{allocIO :: AllocD IO}
\text{allocIO = AllocD allocImp}
\text{where allocImp :: s \rightarrow IO (StateD s IO)}
\text{allocImp v = do r \leftarrow newIORef v}
\text{return (refToStateD r)}
\text{refToStateD :: IORef s \rightarrow StateD s IO}
\text{refToStateD r = StateD (readIORef r) (writeIORef r)}
\]

The idea of dynamically allocating a fresh instance of an effect extends to other kinds of effects. For example, the following $\text{EnvD}$ interface models a read-only environment variable, and a fresh instance can be allocated using an $\text{AllocD}$:

\[
\text{data EnvD r m = EnvD { envM :: m r}}
\]

\[
\text{allocEnvD :: Monad m \Rightarrow AllocD m \rightarrow r \rightarrow m (EnvD r m)}
\]

### 2.5 Local Effects

Using effect polymorphism, functions like $\text{doubleState}$ can be run in the $\text{IO}$ monad, without giving the function access to all possible primitive effects in $\text{IO}$. This removes one of the

\(^4\)Winant and Devriese [38] use a different syntax for dictionary applications: $\text{doubleState @(MonadState Dict getMonadD get_ put_.)}$

\(^5\)Caveat emptor: any efficiency claim in this paper is based purely on our expectations, not yet on benchmarks.
reasons for using classic monads like State, namely restricting the effects that a function can perform. However, we use such monads also to enable effects locally within a restricted scope, but remain purely functional toward outside clients. For example, the purely functional clientPure produces a value of type Int by invoking our previous doubleState example with a local mutable variable:

```haskell
withLocalStatePure ::
  s -> (forall m. MonadState s m => m a) -> a
clientPure :: Int
clientPure = withLocalStatePure 0 client'
  where client' = doubleState >> get
```

To obtain this local mutable state variable, clientPure uses the function withLocalStatePure, which takes an initial value and the effect-polymorphic computation that needs the variable. Internally, withLocalStatePure uses the classic State monad to instantiate the universally quantified m in the argument computation’s type:

```haskell
-- evalState :: State s a -> s -> a
withLocalStatePure v cmd = evalState cmd v
```

In fact, several other monads are designed to offer effects locally. For example, Launchbury and Peyton Jones [23]'s ST monad offers ML-like mutable variables locally. A reference to a mutable variable of type a is represented by an STRef s a and can be allocated and used (read from/written to) inside a monad ST s. The monad comes with a function runST that executes a stateful computation in ST s and returns its result as a pure value.

```haskell
runST :: ∀ a. (∀ s. ST s a) -> a
```

To guarantee that the impurity of the computation cannot be observed from the outside, and that allocated mutable references cannot leak, runST’s type ensures that it can only be applied to functions universally quantified over s.

In our approach, we offer a different API to the same effect:

```haskell
withLocalAlloc ::
  (forall m. Monad m => AllocD m => m a) -> a
```

The idea here is that there’s no reason to expose the user to the ST monad directly. Instead, in the spirit of effect polymorphism, we can just require the computation cmp to be universally quantified over the entire monad it executes in. It just needs to know that this monad supports the MonadAlloc interface. In addition to fitting better into our effect-polymorphism-based approach, this alternative API has the advantage that it doesn’t expose the user to unnecessary detail like the difference between STRef and IORef, which needs to be abstracted from again elsewhere [32].

The implementation of withLocalAlloc is very similar to withLocalStatePure above. We simply instantiate the effect-polymorphic computation in the ST monad, for which we can provide an implementation of AllocD in the same way as for IO before:

```haskell
allocST :: AllocD (ST s)
withLocalAlloc cmp = runST (cmp allocST)
```

Another type of effect we will use is exceptions:

```haskell
class MonadThrow e m where throwM :: e -> m b
```

As for state, we can locally allow the use of exceptions:

```haskell
withLocalThrowPure ::
  (forall m. MonadThrow a m) => m a -> a
```

This function enables the MonadThrow effect locally, for an exception type a equal to the result type of the computation.

A remaining limitation in APIs like withLocalStatePure and withLocalThrowPure is that they only allow us to locally add effects in computations that are otherwise pure. In the next, final section about our approach to effects, we explain how to extend them to locally allow extra effects in computations that are already impure.

2.6 Lifting Effect Interfaces

Imagine, for example, that we use the MonadThrow effect in a function inner, and we want to invoke it from a function outer that should not itself throw exceptions, i.e. all exceptions thrown by inner should be caught inside outer. At the same time, both functions need access to another type of effects: a StatedD Int dictionary representing a mutable variable of type Int:

```haskell
inner :: MonadThrow () m => StatedD Int m => m ()
outer :: Monad m => StatedD Int m => m ()
```

To invoke inner from outer, withLocalThrowPure cannot be used, because it only supports computations of type ∀ m. (Monad m, MonadThrow a m) => m a, i.e. computations that only use exceptions. Instead, we can use the following function withLocalThrow:

```haskell
withLocalThrow ::
  Monad m => (forall n. (Monad n, MonadThrow a n) => LiftD m n a) -> m a
```

As above, withLocalThrow takes a computation running in an arbitrary monad n for which MonadThrow is available.

However, unlike withLocalThrowPure, the monad n does not stand on its own, but is connected to an outer monad m through an interface LiftD m n, which models a monad morphism from m to n:

```haskell
data LiftD m n = LiftD {
  liftDM :: ∀ a. m a -> n a }
```

The method liftDM turns a computation in m to one in n.\(^6\)

With LiftD m n linking the new monad n to the existing monad m, we can now lift existing effects in m into n:

```haskell
liftStateD :: LiftD m n => StatedD s m -> StatedD s n
```

\(^6\)In addition to LiftD, withLocalThrow really also provides an UnliftD m n for lifting more complicated APIs, that do not just produce but also consume computations in m.
liftStateD liftd sd =
  StateD (liftDM liftd (getM sd))
    (\v -> liftDM liftd (putM sd v))

This then enables what we set out to do: invoke inner from within outer by (1) using withLocalThrow to make the MonadThrow effect available locally in a new monad n, (2) lifting the existing StateD effect into n and (3) invoking inner in monad n, with these two effects available:

outer sd = withLocalThrow (\ liftd ->
    inner (liftStateD liftd sd))

While this lifting of effects from monad m into n may seem tedious boilerplate, there are sometimes good reasons to be explicit about lifting effects, for example when an underlying effect can be lifted into the new monad in different ways. For example, AllocD can be lifted in a standard way along a LiftD, but in Section 3.2, we will see an entirely different way to lift an AllocD into a particular monad.

### 3 Factorizing Out the Backtracking Search

Let us now demonstrate our approach for modular effects in practice using μVeriFast: our Haskell-based reimplementation of the VeriFast verifier. However, before we do that in Section 4, this Section first describes an abstraction to capture VeriFast’s backtracking search with angelic and demonic non-determinism in a modular and separately reusable form.

#### 3.1 Angels and Demons in Retreat...

Essentially, this backtracking search can be described in terms of the following type class and the five primitive operations modeled by its methods.

**class MonadAssumeAssert m where**

- `branchDem :: ∀ a. m a -> m a`
- `branchAng :: ∀ a. m a -> m a`
- `failure :: ∀ a. m a`
- `absurdState :: ∀ a. m a`
- `once :: ∀ a. m a -> m a`

First, there are angelic and demonic binary non-deterministic branch operators, modeled by `branchDem` and `branchAng`. Additionally, a `failure` primitive indicates that the search has discovered a failed state. Another primitive `absurdState` indicates that the search has reached a contradictory state that should be considered successful (because unreachable) and not explored further. The latter would, for example, be invoked by μVeriFast when it notices that the current execution point is not reachable (e.g. the then-branch of an `if(false)` statement).

Interestingly, `absurdState` and `failure` are, respectively, neutral elements for `branchDem` and `branchAng`. Intuitively, an always successful branch will never be demonically chosen and an always-failing branch will never be angelically chosen. Finally, the `once` primitive sets a boundary for angelic branching: once a single successful state is reached in a computation `cmp, once cmp` will succeed and forget about any remaining angelic branches within `cmp`, i.e. the backtracking search will not return to those alternative choices even if subsequent computations reach a failure.

In addition to `MonadAssumeAssert`, we provide a default implementation of the backtracking search. It can be used on top of arbitrary underlying effects and can backtrack underlying effects at appropriate times during the search. Backtracking hooks for underlying effects can be provided by instantiating the following interface:

**data BacktrackHooksD m = BacktrackHooksD (BacktrackHooks m)**

- `pushBacktrackBoundM :: m ()`
- `commitM :: m ()`

Three methods need to be instantiated: `pushBacktrackBoundM` registers an additional backtracking boundary, `backtrackM` rolls back effects up to the most recent backtracking boundary and `commitM` drops the most recent backtracking boundary without rolling back any effects.

Our implementation of `MonadAssumeAssert` is made available as a local effect (see Sections 2.5 and 2.6), taking an instance of `BacktrackHooksD m` as a parameter:

**withAssumeAssertSimple :: Monad m => BacktrackHooksD m -> (∀ n. (Monad n, MonadAssumeAssert n) => LiftD m n -> n ()) -> m Bool**

Given backtracking hooks for the underlying monad m, this `withAssumeAssertSimple` will execute a computation in another monad n, for which `MonadAssumeAssert` is implemented. Additionally, like `withLocalThrow` in Section 2.6, the computation can use a `LiftD m n` interface for lifting operations in the underlying monad m into n. Underneath, n will be instanciated with a monad transformer inspired by the `LogicT` monad of Kiselyov and Shan [22]. For space reasons, we do not provide further details about this implementation, but it can be found as part of our implementation.

#### 3.2 Plugging Effects Underneath

Throughout its search of program execution paths, VeriFast incrementally feeds logical assumptions and queries to an underlying SMT solver. When the search backtracks, the solver is told to backtrack its stack of assumptions. μVeriFast achieves this using a `BacktrackHooksD` instance that invokes the appropriate SMT functions:

**smtBacktrackD :: MonadSMT m => BacktrackHooksD m**

μVeriFast currently supports only one underlying SMT solver (Z3 by de Moura and Bjørner [10]), which it interfaces with through a type class `MonadSMT`:

**class MonadSMT m where**

- `assertTerm :: Term -> m ()`
check :: m Bool  
{- ... -}

Combining MonadSMT and MonadAssert, we can implement two pervasive helpers functions assume and assert. The functions invoke the corresponding Z3 operations, but also check satisfiability/provability. In the case of absurd assumptions and unprovable assertions, they cut short the search by unconditionally succeeding resp. failing.\footnote{Note that we use the term assert in the same meaning as C’s assert, i.e. to verify that a certain sanity condition is true.}

-- Make the SMT solver believe that a term is true
assume :: (Monad m, MonadSMT m, MonadAssert m) => Term -> m ()
assume t = do
  assertTerm t
  sat <- check
  unless sat absurdState

isProvable :: (Monad m, MonadSMT m) => Term -> m Bool
-- Verify that a term is provable
assert :: (Monad m, MonadSMT m, MonadAssert m) => Term -> m ()
assert t = do
  prv <- isProvable t
  unless prv failure

A second type of backtrackable effects that we plug underneath the backtracking search is mutable state. To accommodate this, we use a dynamic mutable registry of BacktrackHooks, which provides an interface DynBacktrackD with a single method registerBTHookM:

data DynBacktrackD m = DynBacktrackD { registerBTHookM :: BacktrackHooksD m -> m () }

Using this mutable registry, we provide a backtracking implementation of the AllocD interface that registers appropriate backtrack hooks for every newly allocated mutable variable:

backtrackingAllocD :: Monad m => DynBacktrackD m -> AllocD m -> AllocD m
backtrackingAllocD = (implementation omitted)

In fact, we now have two different implementations of the AllocD interface: for allocating mutable state that is backtracking and non-backtracking respectively. As a result, the µVeriFast code can allocate mutable state variables and choose to use backtracking or non-backtracking ones as appropriate.

4 A Look at the Code

With this tooling in place, we can start building µVeriFast. In this section, we show and explain a few sections of the code that are relevant to the treatment of side effects: interpreting C expressions, C statements, and two symbolic debuggers (a textual and graphical one).

4.1 Interpreting C Expressions

To verify a C program like the one in Figure 1, we need to verify and interpret C expressions like *x != 15, free(y) and (*x)++. These examples already make it clear that C expressions can be effective and include function calls. They are still simpler than C statements though, which may additionally contain control flow primitives like return or break.

Still, interpreting expressions is far from trivial. For example, interpreting the expression *x != 15 is only valid when the current environment maps x to a logical term t, the symbolic heap contains a predicate t₁ → t₂ and the SMT solver confirms that t₁ = t. The result of the interpretation value will depend on the term t₂. The expression (*x)++ does not even just inspect the symbolic heap, but also modifies it, and expressions like y += 10 or z = 3 modify the environment. Interpreting a call like free(y) is only possible if we know the contract that has been declared for free and all of the above expressions of course need to be able to fail and report errors. In other words, interpreting an expression may produce side effects: writing to the current environment and symbolic heap, reading contracts and reporting errors.

So, let us make interfaces for these effects available to the function interpExpr. We use the following data types:

data Pred = PointsTo Term Term

 type SymHeap = [Pred]

 type Environment = Map Ident Term

Atomic separation logic predicates are represented in the type Pred. Pred only models simple points-to predicates x → y, where x and y are SMT terms Term. Symbolic heaps (SymHeap) are simply lists of atomic predicates with SMT terms Term. Finally, environments Environment map variable identifiers of type Ident to SMT terms Term.

The effects we need in interpExpr are then captured by the following type classes:

class MonadLog String m => MonadAnalysis m where
  errorM :: Y a. CodeLocation -> String -> m a

class MonadAnalysis m, MonadSMT m, MonadAssume m => MonadSepC m where
  contractsS :: StateD (Map Ident Contract) m

class (MonadAnalysis m, MonadSMT m, MonadAssume m) => MonadSepC m where
  symHeapS :: StateD SymHeap m

impEnvS :: StateD Environment m

The MonadAnalysis method errorM signals a verification error, and the contractsS state dictionary gives access to a registry of declared contracts. MonadSepC provides state dictionaries for the symbolic heap and environment.

The function interpExpr is then defined as an effect-polymorphic function with the constraint MonadSepC m. Please ignore the MonadDebug m constraint for now; we will come back to it in Section 4.4.
The function takes a C expression in a representation from the language-c library\(^8\), which we use for parsing and typechecking C code. It interacts with the side effects mentioned above and returns the result as an SMT term `Term`. Additionally, for C expressions which are l-values (i.e. values that may be used as assignee in an assignment), `interpExpr` returns an `LValue` which we use elsewhere to interpret assignments.

The function is defined by case analysis on the expression AST `CExpr`. Let us look at some of the cases, to see how the side effects are produced. Constant expressions are interpreted trivially, without producing any effects:

```haskell
interpExpr (CConst cnst) =
  return (Nothing, interpConstant cnst)
```

More interesting is the interpretation of variables, where we make use of the environment of local variable interpretations that are available as a mutable state variable:

```haskell
interpExpr (CVar x info) =
  do env <- getM impEnvS
     case Map.lookup x env of
     Just t -> return (Just (LVar x), t)
     Nothing -> errorM info "Var not found"
```

We get the current environment using the `impEnvS` dictionary of type `StateD Environment m` that we have access to through `MonadSepC m`. We then simply return the interpretation of the variable, if any, and fail otherwise.

Also interesting are pointer dereferencing expressions `*e`:

```haskell
interpExpr (CUnary op e info) =
  do (lv, t) <- interpExpr e
     interpUnaryOp info op (lv, t)
  return (PointsTo lv, t)
```

`interpUnaryOp info op (lv, t)` will first recursively interpret e to a logical variable `t` and then interpret the indirection operator `*` using a second function `interpUnaryOp`. That function will assert the presence of an atomic separation logic predicate `t \rightarrow val`, using the function `assertPred`\(^9\). The latter is returned as the interpretation result, along with an appropriate `LValue`.

The function `assertPred` will first non-deterministically take an arbitrary predicate from the symbolic heap using a function `takePred`, match it against the required arguments and return the result. The non-determinism used is angelic, so that it is sufficient if one of the chosen predicates makes the subsequent matches succeed. As a final example of the use of side effect interfaces, we take a closer look at `takePred`:

\(^9\)http://hackage.haskell.org/package/language-c

\(^8\)The `DummyPat` is a remnant of a pattern matching system that we have simplified away.

The Haskell code for the `MonadSepC m` and `MonadDebug m` monads is:

```haskell
data MonadPlusD m = (implementation omitted)
selct :: [a] -> ([a], [a])
chooseM :: Monad m => MonadPlusD m -> [a] -> m a
angelicChoice, demonicChoice ::
  MonadAssumeAssert m => MonadPlusD m

takePred :: (Monad m, MonadSepC m, MonadDebug m) =>
  CExpr -> m (Maybe LValue, Term)
takePred = do
eval <- getM symHeapS
  (p, heap') <-
  chooseM angelicChoice
takePred = do
  (lv, t) <-
  putM symHeapS heap'
  return lv
```

This `takePred` function gets the symbolic heap using the dictionary `symHeapS` that is available through the constraint `MonadSepC m`. It then uses function `select` to split the list into a single atomic predicate and the remaining ones, in all possible ways, and then uses the function `chooseM` to nondeterministically choose one. To do this, `chooseM` requires a dictionary of type `MonadPlusD`. Through the constraint `MonadAssumeAssert m`, two implementations of this dictionary are available: `angelicChoice` and `demonicChoice`, and we choose the former. Next, `takePred` updates the symbolic heap to remove the chosen predicate and returns it.

We hope the reader agrees that the above code and its implementations are elegant and not that hard to follow. Interactions with side effects are only apparent in those functions that directly use them and they are kept nicely abstract. Functions with effects are defined in an arbitrary monad `m` (i.e. they are effect-polymorphic), but otherwise look quite standard.

### 4.2 Function Invocations

Another interesting case is the verification of function invocations, in the function `interpFunCall`:

```haskell
localSM :: Monad m =>
  StateD r m r -> m a -> m a
getSM :: Monad m => StateD s m a -> m a

interpExpr (CCall CVar f _ info) =
  do res <- interpFunCall info f args
     return (Nothing, res)
```

The Haskell code for the `Monad SepC m`, `MonadDebug m` and `MonadPlusD m` monads is:

```
data MonadPlusD m = (implementation omitted)
select :: [a] -> ([a], [a])
chooseM :: Monad m => MonadPlusD m -> [a] -> m a
angelicChoice, demonicChoice ::
  MonadAssumeAssert m => MonadPlusD m

takePred :: (Monad m, MonadSepC m, MonadDebug m) =>
  CExpr -> m (Maybe LValue, Term)
takePred = do
eval <- getM symHeapS
  (p, heap') <-
  chooseM angelicChoice
takePred = do
  (lv, t) <-
  putM symHeapS heap'
  return lv
```

This `takePred` function gets the symbolic heap using the dictionary `symHeapS` that is available through the constraint `MonadSepC m`. It then uses function `select` to split the list into a single atomic predicate and the remaining ones, in all possible ways, and then uses the function `chooseM` to nondeterministically choose one. To do this, `chooseM` requires a dictionary of type `MonadPlusD`. Through the constraint `MonadAssumeAssert m`, two implementations of this dictionary are available: `angelicChoice` and `demonicChoice`, and we choose the former. Next, `takePred` updates the symbolic heap to remove the chosen predicate and returns it.

We hope the reader agrees that the above code and its implementations are elegant and not that hard to follow. Interactions with side effects are only apparent in those functions that directly use them and they are kept nicely abstract. Functions with effects are defined in an arbitrary monad `m` (i.e. they are effect-polymorphic), but otherwise look quite standard.

The Haskell code for the `Monad SepC m`, `MonadDebug m` and `MonadPlusD m` monads is:

```
data MonadPlusD m = (implementation omitted)
select :: [a] -> ([a], [a])
chooseM :: Monad m => MonadPlusD m -> [a] -> m a
angelicChoice, demonicChoice ::
  MonadAssumeAssert m => MonadPlusD m
```
We do (1) using contracts.

To demonstrate how we can specify and pass precisely the functional expression’s result: branches (with appropriate assumptions about the condition and demonically branch between the then and else automatically ignoring the unnecessary MonadCControl transparently filled in by the ones available for interpStmt.

Notice how the effect interfaces required by interpStmt are implemented by other components. For example, here’s an excerpt of how we invoke interpStmt from the function interpFunDef that verifies a top-level function declaration:

```haskell
interpFunDef :: (Monad n, MonadSepC n, MonadDebug n) => n ()
interpFun = do _ <- produceAsn pre
  withLocalThrow (\ liftd unliftd ->
    interpBody ((liftMonadCControl liftd unliftd)
      getMonadCControlDict))
  _ <- consumeAsn post
  leakCheck

interpBody :: (Monad n, MonadSepC n, MonadDebug n, MonadThrow () n) => n ()
interpBody = interpStmt ((ccont)) stmt
where returnK ret = do assignVar "result" ret
  throwE ()
  notInLoopE = errorM "not in loop!"
  gotoK l = (implementation omitted)
  ccont = MonadCControl.Dict gotoK
  notInLoopE notInLoopE returnK
```

In the case analysis of interpStmt, the control effect interpretations from MonadCControl are used for the appropriate C statements:

```haskell
interpStmt :: MonadCControl m where
labelsM :: Id -> m ()
continueM :: m ()
breakM :: m ()
returnM :: Term -> m ()

interpStmt ::
  (Monad m, MonadSepC m, MonadCControl m, MonadDebug m) => CStat -> m ()

The implementation of interpStmt is another big case analysis of the statement at hand. Statements that are just expressions are simply passed to interpExpr:

```haskell
interpStmt (CExpr e) = do _ <- interpExpr e
  return ()
```

Notice how the effect interfaces required by interpStmt are transparently filled in by the ones available for interpStmt, automatically ignoring the unnecessary MonadCControl.

For an if-statement, we interpret the conditional expression and demonically branch between the then and else branches (with appropriate assumptions about the conditional expression’s result):

```haskell
consumeAsn pre
res <- freshValue "result"
localSM impEnvS (Map.insert "result" res)
(produceAsn post)
return res
where buildEnv = ...
```

Interpreting a function call in VeriFast means (1) looking up the function’s contract, (2) consuming the function’s precondition and (3) producing its postcondition. In interpFunCall, we do (1) using contractsS of type StateD (Map Ident Contract) m: the mutable reference to the repository of function contracts that we have access to through the MonadSepC constraint. To perform (2) and (3), we must not interpret the pre-and postcondition under the environment that is active during the function call, but under an environment where the pre- and postcondition’s free variables are defined appropriately. Those free variables are the function’s arguments and the values for them (argvVals) can be found by interpreting the argument expressions of the invocation. We temporarily activate this environment to consume the precondition using localSM. The postcondition has an additional result variable in scope, which we postulate as a fresh logical variable, add to the environment, and return as the function call’s result after producing the function’s postcondition.

### 4.3 Interpreting C Control Flow

To demonstrate how we can specify and pass precisely the effect interfaces we need, let us take a look at verifying C statements. The main difference with expressions is that we need to deal with C control effects: goto, continue, break and return. The function interpStmt requires interpretations for these effects in an effect interface MonadCControl:

```haskell
class MonadCControl m where
labelsM :: Id -> m ()
continueM :: m ()
breakM :: m ()
returnM :: Term -> m ()

interpStmt ::
  (Monad m, MonadSepC m, MonadCControl m, MonadDebug m) => CStat -> m ()
```

The implementation of interpStmt is another big case analysis of the statement at hand. Statements that are just expressions are simply passed to interpExpr:

```haskell
interpStmt (CIf e sl s2) =
  do _ <- interpExpr e
    branchDem (assume t >> interpStmt sl)
    (assume (Not t) >> interpStmt s2)
```

Finally, what’s interesting is how these effects can be implemented by other components. For example, here’s an excerpt of how we invoke interpStmt from the function interpFunDef that verifies a top-level function declaration:

```haskell
 interpFunDef :: (Monad n, MonadSepC n, MonadDebug n) => n ()
 interpFun =
  do mapM_ instantiateParam params _ <- produceAsn pre
    withLocalThrow (\ liftd unliftd ->
      interpBody ((liftMonadSepC liftd unliftd)
        getMonadSepCDict))
    _ <- consumeAsn post
    leakCheck
```
When VeriFast complains about missing separation logic with only minimal modifications to the interpretation of when a return statement is reached. Concretely, the verification or abort. Adding this feature to µ:: LiftD

:: MonadDebug

:: Monad

be defined to just ignore every invocation of the interpStmt

tiating the MonadDebug

m

class MonadDebug

m

then been reached.

With the MonadThrow interface that is available inside interpBody, we can then use the throwM effect to escape when a return statement is reached. Concretely, the MonadCControl

ccont deals with return this way (after assigning the return value to the variable result). It throws errors for breaks (since we’re not in a loop yet) and deals with gotos in a way that we don’t go into.

4.4 A Console Debugger

When VeriFast complains about missing separation logic assertions (for example, to satisfy the assumptions of a function call), it can be useful to inspect or manipulate the state of the symbolic heap, environment or path condition at a specific location in the code. VeriFast accomodates this with a symbolic debugger. The programmer can put a breakpoint in the code, inspect state, single-step the program, continue verification or abort. Adding this feature to µVeriFast is a nice example of modular effects, and in this section and the next, we discuss how to add a console and a GUI debugger with only minimal modifications to the interpretation of expressions and statements.

We start by adding a MonadDebug type class which defines a hook beforeStmtM. The only modification we make to already-discussed code is that we invoke this hook before executing any statement, to check whether a breakpoint has been reached.

class MonadDebug m where

beforeStmtM :: CodeLocation -> m ()

interpStmt :: (Monad m, MonadSepC m, MonadCControl m, MonadDebug m) => CStat -> m ()

interpStmt s = do beforeStmtM (annotation s)

interpStmt' :: (Monad m, MonadSepC m, MonadCControl m, MonadDebug m) => CStat -> m ()

interpStmt' = (cases shown before)

Implementing a debugger then amounts simply to instan-
tiating the MonadDebug constraint for the monad in which interpStmt is invoked. For example, a trivial debugger can be defined to just ignore every invocation of the beforeStmtM hook:

noopMonadDebug :: Monad m => MonadDebug Dict m

noopMonadDebug =

MonadDebug Dict (\ _ -> return ())

For a console front-end of µVeriFast, we can implement a more interesting debugger consoleMonadDebug, which gives the user a Read-Eval-Print Loop (REPL) for inspecting µVeriFast’s state:

cameraMonadDebug ::

(Monad m, MonadSepC m) =>

LiftD IO m -> StateD [Int] m ->

MonadDebug Dict m

consoleMonadDebug lio bpsd =

consoleMonadDebug

m n

where

beforeStmtM :: CodeLocation -> m ()

beforeStmtM info =

let
d

d = do bpts <- getM bpsd

let
d = do liftDM lio (putStrLn "$")

cmd <- liftDM lio getLine

exec cmd

d = do debug :: m ()

d = do debug = do liftDM lio (putStrLn "$")

cmd <- liftDM lio getLine

exec cmd

d = exec :: String -> m ()

d = exec "fail" = failure

d = exec "continue" = return ()

d = exec "heap" = printHeap >> do debug

d = exec _ = do liftDM lio (putStrLn "$")

do debug

This console debugger takes a mutable reference bpsd of type StateD [Int] m to the currently registered breakpoints (i.e. a list of line numbers). When the MonadDebug hook is invoked, it checks whether the line number of the current statement is in this list and if so, starts the debugger REPL. This loop is then implemented by using the MonadSepC m effects interface, together with a LiftD IO m interface that lets us produce arbitrary external IO effects (and particularly interact with the console).

4.5 A Graphical Debugger

Console interfaces are not to everyone’s liking, so we also implement a GUI debugger as part of µVeriFast’s GUI front-end. This is a bit harder because that front-end is an asyn-
chronous application. When verification stops at a breakpoint, we cannot just block for user input and continue verification when it arrives. Instead, when a breakpoint is reached, we have to suspend the verification, store its state and update the GUI to reflect the interrupted state. When user input arrives through the invocation of a button press handler, we must then resume verification in the captured state.

This calls for a form of delimited continuations [13], captured by the following MonadShift type class:

class MonadShift r m n

where

shiftM :: ((a -> m r) -> m r) -> n a
We cannot give a thorough explanation of delimited continuations for space reasons, but essentially, `MonadShift m n` expresses that computations in monad `n` execute in a delimited continuation scope where continuations live in monad `m` and have return type `r`. To produce a computation of type `n a, shiftM :: ((a -> m r) -> m r) -> n a` allows us to suspend and reify the continuation of type `a -> m r` and return a result of type `m r` directly. The continuation of type `a -> m r` may be used zero, one or multiple times.

For our GUI debugger, we put the continuation delimiter around the verification of a source file, and its result will be of the following `VerifyResult` type:

```haskell
data VerifyResult m = VSuccess
                   | VFail CodeLocation String
                   | VPaused (DebugIntf m)
```

That is: we return whether the verification was successful, failed (at a given source location with a given error message), or paused (i.e. a breakpoint was hit). In the latter case, we return a value of type `DebugIntf m` that allows us to continue the verification.

In terms of this interface, we implement our GUI debugger. The function `guiDebugger` takes a mutable reference to the currently registered breakpoints (as a list of line numbers) and returns a dictionary for the `MonadDebug` type class.

```haskell
guiDebugger ::
  (Monad m, Monad n, MonadSepC n, MonadShift (VerifyResult m) m n) =>
  StateD [Int] n -> MonadDebug.Dict n
```

In the `beforeStmtM` hook, we again check whether the line number is in the list of breakpoints. If so, we use `shiftM` to get the current continuation, and return a `VPaused` value that invokes this continuation in the continue callback. This callback is then appropriately invoked by the GUI.

### 5.1 Modular Effects

Our implementation of μVeriFast demonstrates the value of modular effects. The code described in Section 4 is implemented in terms of abstract effectful interfaces, and remains oblivious to the precise way that VeriFast’s backtracking search or other effects are implemented. In fact, the code was entirely unaffected when we started to run it in a continuation monad for the GUI debugger, and if we wanted, there would be no problem using the guiDebugger and consoleDebugger together in the same application. It would be interesting to implement μVeriFast using alternative modular effect libraries (see Section 6.3) and use μVeriFast’s advanced effect interactions as a benchmark to compare them.

### 5.2 Dictionary Applications

As the DictionaryApplications extension by Winant and Devriese [38] is still new and non-final, it is useful to evaluate our experience with it. Our use of explicit dictionary applications for effect interfaces demonstrates that the extension effectively enables important new Haskell design patterns. Technically, some lessons can be learned from our use of the extension, related to the conditions which Winant and Devriese require on explicit dictionary applications (to preserve the properties of global instance uniqueness (GIU) and coherence). We ran into a number of cases where the global uniqueness criterion was overly restrictive and although we were able to work around the restrictions (sometimes by just using explicit dictionaries instead), we suggest sound relaxations of the criterion.

**Higher-order roles** A restriction in the non-higher-order role system of GHC [7] leads to an overly conservative inference for the role of type variables `m` that are used within an argument of another type variable. An example is the argument `m` of the class `MonadAlloc` in Section 2.4, because of how `m` is used in the type `alloc :: s -> m (StateD s m)`. This may be solvable by building on the QuantifiedClassConstraints GHC extension [6, 29].

**Simultaneous dictionary applications** The current GIU criterion could deal better with cases where a Haskell function is explicitly applied to multiple dictionaries simultaneously, for example in `interpBody` from Section 4.3. When regarding these applications separately, the second is rejected, but DictionaryApplications could accept them together without compromising soundness.

**Applying a dictionary for a class but not its parent** Finally, type class constraints can currently only be instantiated together with all of their superclasses. It would be useful for the extension to support instantiating a subclass but not its parent.

There are also some questions to be looked at in more detail. For example, it is not clear how DictionaryApplications would work for classes with FunctionalDependencies.
We believe this paper provides strong evidence for the usefulness of the DictionaryApplications extension, and hopefully strengthens the case for working it out further.

5.3 Effect Polymorphism

Compared to other approaches to modular effects, our use of effect polymorphism brings two unique advantages.

First, effect-polymorphic types like \( \forall \ m \rightarrow (\text{Monad} \ m, \text{MonadState} \ m) \rightarrow m \text{Bool} \) are not that hard to understand. They only make use of two very common Haskell features: parametric polymorphism and monads. To fully understand the types that are at play in other approaches [20], it is necessary to understand non-trivial category-theoretical concepts like free monads and left Kan extensions.

Second, effect-polymorphism in a parametrically polymorphic language like Haskell automatically implies parametricity results, i.e. free theorems [37]. This effect parametricity implies interesting results about how computations remain independent of the implementation of effect interfaces and has already been studied by Voigtländer [36].

There are also deep connections with the semantics of object-oriented languages. Devriese et al. [11] have used effect parametricity at a semantic level to formalise capability safety. This language property characterises object-capability languages, a type of programming language with important security applications [25].

6 Related Work

In addition to the related work already mentioned in previous sections, we relate our work to other approaches to modular effects in Haskell.

First, as mentioned in Section 2.1, there are two additional ways to work around the lack of local instances in Haskell, when applying effect polymorphism: the ReaderT design pattern and a technique known as reflection. In sections 6.1 and 6.2, we explain them briefly and the advantages of our use of dictionary applications, compared to them. Essentially, this will show that our approach is the first to enable full use of effect polymorphism in Haskell, without unnecessary syntactic and runtime overhead.

In Sections 6.3 and 6.4, we look at more distant related work based on algebraic effects and free monads, and other ideas.

6.1 The ReaderT Design Pattern

The first alternative is known as the ReaderT design pattern [31]. Essentially, computations are also formulated as effect polymorphic functions with effect interfaces like our doubleState from Section 2. However the effects are instantiated differently:

```haskell
instance MonadState Int
    (ReaderT (IORef Int) IO) where
  get = do x <- ask
  put v = (implementation omitted)
```

```haskell
client :: IO ()
client = do r <- newIORef (2 :: Int)
  runReaderT doubleState r
```

Computations are not executed in a monad \( m \), but in ReaderT env where env is a type that contains all the data necessary for implementing the effect interfaces.

Essentially, we think the ReaderT pattern can be applied to the same effect as our DictionaryApplications. However, it is more difficult to understand, requires more boilerplate code and sometimes the use of UndecidableInstances. Some (but not all) boilerplate can be avoided using the recent DerivingVia extension [5, 14]. While a dictionary application directly desugars to a single function application, the ReaderT will add a function application to every monadic bind in effect-polymorphic functions, leading to a possible performance overhead, unless the compiler is able to re-eliminate those applications.

6.2 Reflection

A second way to bypass the lack of local instances in Haskell, is based on the reflection library, designed after a proposal by Kiselyov and Shan [22]. We do not go into details for space reasons, but this approach also allows to invoke a function like doubleState with an arbitrary MonadState instance containing local values. However, it is technically complex and requires additional boilerplate like spurious newtype definitions and instances. When more than a single value is needed in the definition of a (set of) local type instances, the syntactic overhead is even larger. Additionally, defining the instances for the newtype wrappers sometimes requires UndecidableInstances (or technical tricks to avoid it).

6.3 Algebraic Effects and Handlers

Algebraic effects and handlers [3, 26], another approach to modular effects, offer a way to define and combine effect interfaces, to define functions that use them and to invoke those functions with a specific implementation of the effects (handlers). The approach has been implemented in Haskell libraries by Kammar et al. [19] and Kiselyov et al. [21].

In these frameworks, effectful functions are not implemented in an arbitrary monad that offers the necessary effect interfaces (like in our approach), but in a kind of universal monad: the free monad, parametrised by the effects available. This free monad is encoded differently in different frameworks, often using category-theoretical concepts.

A problem with this approach is that instantiating effects in several steps leads to several encodings and decodings of the free monad, and ultimately suboptimal performance, although people have worked to fix this [39]. Our approach does not require changing the representation of effectful code.
just for instantiating effect interfaces and as such, avoids this source of suboptimal performance.

It is worth noting that certain problems related to proper scoping of effect handlers [4, 40] simply do not arise in our approach, essentially because we pass around effect implementations through already well-scoped function arguments and type classes, rather than build a separate mechanism for dispatching effects. This contrasts, for example, with the approach by Kiselyov et al. [21], which problematically relies on runtime type information to dispatch effects.

6.4 Other Related Work

Jaskelioff [18] describes the effects library Monatron, which uses a form of explicit dictionaries, similar to what we described in Section 2.2. He combines an explicit dictionary like our StateD s m with a companion type class StateM s m like the following:

```haskell
class StateM s m where stateM :: StateD s m
```

Compared to our approach, this means an effectful function must either take an explicit StateD dictionary argument, allowing for non-unique and local implementations of the effect, at the cost of extra bookkeeping to pass around the explicit dictionary, or use a StateM constraint with no bookkeeping, but no support for non-unique or local implementations. Jaskelioff also defines a uniform way to lift dictionaries like StateD s m to a transformed monad t m, which would be useful for us too, especially if it can support the lifting over arbitrary monad morphisms supported by us and the monatron theory [17].

Schrijvers and Oliveira [28] also use monad morphisms as part of a framework for working with monad transformer stacks, albeit for quite different purposes.

7 Conclusion

This paper combines effect polymorphism with the recently proposed DictionaryApplications extension to obtain a new, general way to deal modularly with effects in Haskell. We have demonstrated the power of the approach using the case study of reimplementing VeriFast. The resulting code is much cleaner and more modular, providing evidence for the importance of modular effects in real applications, but also for the practicality of our approach at modular effects. As a side result, we hope our results encourage someone to nurture the DictionaryApplications GHC extension to maturity.

Acknowledgments

We thank Bart Jacobs and Andrey Mokhov for their comments and suggestions on this paper.

References


