

Programming Monads Operationally with Unimo

Chuan-kai Lin

Department of Computer Science
Portland State University
cklin@cs.pdx.edu

Abstract

Monads are widely used in Haskell for modeling computational effects, but defining monads remains a daunting challenge. Since every part of a monad’s definition depends on its computational effects, programmers cannot leverage the common behavior of all monads easily and thus must build from scratch each monad that models a new computational effect.

I propose the Unimo framework which allows programmers to define monads and monad transformers in a modular manner. Unimo contains a heavily parameterized observer function which enforces the monad laws, and programmers define a monad by invoking the observer function with arguments that specify the computational effects of the monad. Since Unimo provides the common behavior of all monads in a reusable form, programmers no longer need to rebuild the semantic boilerplate for each monad and can instead focus on the more interesting and rewarding task of modeling the desired computational effects.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.13 [*Software Engineering*]: Reusable Software; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Functional constructs

General Terms Design, Languages

Keywords Haskell, monads, monad transformers, Unimo

1. Introduction

Monads are a mathematical abstraction that can be used to model a wide variety of computational effects. They were initially used by Moggi to structure denotational semantics [16, 17] and were later introduced into Haskell to support input–output operations [7]. In Haskell, monadic computations are first-class values which can be stored in data structures, and programmers can define their own monad to model computational effects including state, exceptions, nondeterminism, resumptions, and continuations. These qualities made Jones conclude that “Haskell is the world’s finest imperative programming language” [6].

While monads with multiple effects can be built using monad transformers, composing existing monad transformers may not be sufficient to satisfy all application-specific needs. Unfortunately, defining a monad is an undertaking not for the faint of heart. After formulating the desired computational effects and how they should be made available to the users of the monad, the programmer must come up with a monad type constructor that represents the effects, define monad operators on the monad type, implement the programming interface to the effects, and verify that the operators satisfy a set of algebraic properties called the monad laws. If the computational effects need to be made available on top of another monad, the programmer must instead define a monad transformer, which is even more complicated because effects in the underlying monad may interact with those in the monad transformer in some unexpected manner.

I propose a monadic programming framework called *Unimo* which simplifies the definitions of monads and monad transformers. Unimo eliminates the need to define a monad type constructor in accordance with the computational effects, and it consolidates the semantics of a monad into an evaluation function. Evaluation transitions on monadic computations enforce the monad laws by construction, so programmers defining a monad need only to focus on the computational effects. Finally, Unimo highlights the deep connections between monads and monad transformers, and it leads to a procedure for adapting a Unimo-based monad into the corresponding monad transformer. The implementation of Unimo uses rank-2 types [9], generalized algebraic datatypes [10], and lexically-scoped type variables [8]; thus it works only with recent versions of GHC. I believe that Unimo makes it easier not only to define monads, but also to define monads *correctly*. The specific technical contributions of this paper are:

- A technique for defining monads using operations instead of denotations (Section 3)
- A *parameterized term representation* which can capture computations in any monad (Section 4.1)
- An *observer function* which implements the common behavior of monads and enforces the monad laws (Section 4.2)
- An extension of the Unimo framework that supports monad transformers (Section 6.3)
- A procedure for adapting Unimo-based monads into the corresponding monad transformers (Section 6.4)
- A technique for meta-programming on Unimo-based monadic computations (Section 7)

This paper also contains the following examples:

- An implementation of the continuation monad that is not based on continuation passing (Section 4.3)

```

instance Monad [] where
  return x    = [x]
  [] >>= _    = []
  (x:xs) >>= k = k x ++ (xs >>= k)

class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

instance MonadPlus [] where
  mzero = []
  mplus u v = u ++ v

```

Figure 1. The list monad in Haskell. The Monad instance declaration overloads the monad operators, and the MonadPlus instance declaration defines the effect basis.

- An implementation of Claessen’s parallel parsing processes [1] constructed from an algorithm description instead of through formal derivations (Section 5)
- An implementation of the list monad transformer that does not require a commutative underlying monad (Section 6.4)
- Two functions that implement transparent tracing of monad and monad transformer computations (Section 7.2)
- A sandboxing function that denies all file open requests in an IO computation (Section 7.3)

Section 2 presents background information on monads, Section 8 discusses related work, and finally Section 9 concludes the paper.

2. Monadic programming in Haskell

In Haskell, a monad consists of a type constructor $M :: * \rightarrow *$ and two monad operators with the following types:

```

return :: a -> M a
(>>=)  :: M a -> (a -> M b) -> M b

```

A value of type $M a$ is a monadic computation in the monad M that produces a result of type a . Intuitively, $\text{return } v$ is a computation that produces the value v without introducing any computational effects, and $m \gg= k$ (read “ m bind k ”) is a computation that runs m , applies k to the result of m to produce another computation n , and then runs n . The intuition is formally characterized as a set of algebraic properties called the *monad laws* as follows:

- L1. Left unit law: $\text{return } v \gg= k \equiv k v$
- L2. Right unit law: $m \gg= \text{return} \equiv m$
- L3. Associativity law: if v is free in k and g , then $(m \gg= k) \gg= g \equiv m \gg= (\lambda v \rightarrow k v \gg= g)$

In Haskell the operators are declared in the Monad type class, and programmers should overload them when defining a monad.¹

Figure 1 shows a definition of the list monad in Haskell. The monad models nondeterministic computations which can produce multiple results. The monad type constructor of the list monad is the list; the return operator produces a single-element list, and the $\gg=$ operator applies the continuation k to each result of the first computation and then concatenates the results of the applications. The MonadPlus type class declares two computations that allow programmers to invoke the nondeterminism effect of the monad: mzero represents a computation with no results, and mplus merges

¹The Monad type class also includes a fail computation, but I will not consider it here because it is not part of the mathematical formulation of a monad.

```

newtype State s a =
  State { runState :: s -> (a, s) }

instance Monad (State s) where
  return v = State (\s -> (v, s))
  m >>= k = State $ \s ->
    let (v, s1) = runState m s
    in runState (k v) s1

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

instance MonadState s (State s) where
  get = State (\s -> (s, s))
  put s = State (\_ -> ((), s))

```

Figure 2. The state monad in Haskell. The type constructor of the monad is State which defines a state transformer. The $\gg=$ operator works by passing the state $s1$ produced by m to the continuation k . The get computation returns the current state, and the put computation sets the state.

the results of two computations. The list monad defines mzero as the empty list, and mplus as the list concatenation function. The idea behind MonadPlus is that even though there are many list monad computations that make use of the nondeterminism effect, mzero and mplus are really all that programmers need to take full advantage of the list monad. I will refer to such computations as the *effect basis* of a monad.

Figure 2 shows a definition of the state monad, which models stateful computations, in Haskell. The monad type constructor is a state transformer function parameterized by the type of the state s , the return operator produces a computation that leaves the state unchanged, and the $\gg=$ operator composes the state transformer in m with the uncurried version of the continuation k . (The actual definition of $\gg=$ is a little more complicated because it needs to deal with the State data constructor.) The effect basis of the state monad consists of two computations: get , which retrieves the current state, and $\text{put } s$, which changes the current state to s .

Haskell programmers define monads through denotations (what a thing is). The monad type constructor denotes what a monadic computation is, and various Haskell expressions denote what the monad operators and the effect basis are. While denotations are a useful and well-established technique for defining the semantics of programming constructs [18], I believe that they unnecessarily complicate the definition of monads for the following reasons:

Monad laws The Haskell compiler ensures only that the return and $\gg=$ operators in an instance of the Monad type class have the right types, and it leaves open the possibility that the operators may not satisfy the monad laws. While it is in principle possible to constrain the denotations of the monad operators so that they must satisfy the monad laws, such a strategy is difficult to implement in practice because nobody had come up with an appropriate set of constraints, and because there is no way to enforce any such constraints at compile time. The issue of restricting the denotations to ensure the compliance with monad laws is analogous to achieving full completeness in denotational semantics, which is generally considered a hard problem.

Separation of concerns Given that programmers need to shoulder the responsibility of ensuring compliance with monad laws, we would hope for a clean separation between the monad laws and the computational effects modeled by the monad. Since the monad laws depend only on the interaction between monad operators, and the computational effects depend only on the interaction within the effect basis, such a hope is not really all that unreasonable.

```

data Plus a
  = Unit a
  | forall b. Bind (Plus b) (b → Plus a)
  | Zero
  | Plus (Plus a) (Plus a)
instance Monad Plus where
  return = Unit
  (≫=) = Bind
instance MonadPlus Plus where
  mzero = Zero
  mplus = Plus
run_list :: Plus a → [a]
run_list (Unit a)           = [a]
run_list m@Zero            = run_list (Bind m Unit)
run_list m@(Plus _ _)     = run_list (Bind m Unit)
run_list (Bind (Unit v) k) = run_list (k v)
run_list (Bind (Bind m k) g) = run_list (Bind m cont)
  where cont v = Bind (k v) g
run_list (Bind Zero _)     = []
run_list (Bind (Plus m n) k) = ms ++ ns
  where ms = run_list (Bind m k)
        ns = run_list (Bind n k)

```

Figure 3. An operational definition of the list monad. Instead of defining the denotation of list computations directly, here I capture a list computation as algebraic data with the term representation `Plus` and evaluate the captured computation with the evaluation function `run_list`.

Defining monads through denotations hinders such separation of concerns because the denotations of different parts of a monad are highly interrelated. Modeling a computational effect requires changing the denotation of computations, which in turn affects the denotations of monad operators. Likewise, changing the denotation of the $\gg=$ operator affects the interaction within the effect basis. This tight coupling makes it difficult to investigate and to change one aspect of a monad without causing unintended consequences elsewhere.

Operational characteristics The denotational perspective alone is inadequate because it focuses only on *what* result should be produced by a computation and not on *how* the result should be produced. In addition to correctness, practical programmers also care about the space and time efficiency of a monad [1, 4], and working with denotations makes it difficult to investigate or to tweak these operational characteristics.

In Section 3 I will show how defining monads through operations (what a thing does) complements the denotational perspective and addresses the aforementioned issues.

3. Defining monads with operations

The basic strategy for specifying an operational semantics of a programming language is to define how evaluation should proceed for each valid program state. Using the list monad as an example, I now illustrate how to apply the same strategy to defining monads with operations.

3.1 Monad term representation

An operational definition of a monad consists of two parts: a *term representation* to capture a monadic computation as algebraic data, and an *evaluation function* to evaluate the captured computation to its result. For example, the `Plus` datatype in Figure 3 is a term rep-

resentation of the list monad. The `Unit` and `Bind` data constructors represent the monad operators, and `Zero` and `Plus` represent the `mzero` and `mplus` computations in the effect basis. The `Plus` type is a trivial instance of the `Monad` and the `MonadPlus` type classes. Data constructors in a term representation act as placeholders for monad operators and the effect basis; they do not implement any computational effects but merely capture a monad computation as an algebraic data value.

3.2 Monad evaluation function

For the list monad to be useful, we need to provide a way to turn a computation into its results. Following the strategy for specifying operational semantics, I use an evaluation function to define the evaluation transitions on monadic computations. There should be one transition for each valid monadic computation, and if the evaluation is stateful, the transitions should be defined over all valid state-computation combinations.

The `run_list` function in Figure 3 evaluates list monad computations. It performs a case analysis to determine the top-level structure of the computation and evaluates the computation using the corresponding evaluation transition. The cases and the transitions in `run_list` are:

1. `Unit a`: evaluate to a single-element list
2. `Zero`: apply the right-unit monad law
3. `Plus _ _`: apply the right-unit monad law
4. `Bind (Unit v) k`: apply the left-unit monad law
5. `Bind (Bind m k) g`: apply the associativity law
6. `Bind Zero _`: evaluate to the empty list
7. `Bind (Plus m n) k`: create two sub-computations by distributing `Plus` over `Bind` and then concatenate the results `ms` and `ns` of evaluating the sub-computations

These seven cases are exhaustive and the transitions fully specify the evaluation of list monad computations. Note that the effect basis plays very different roles in the denotational and operational definitions of a monad. In the denotational definition (see Figure 1 for an example), the computational effects are defined through the monad operators, and the effect basis is there only to help programmers use the monad. In the operational definition, the computational effects are defined in terms of the interaction within the effect basis, so the effect basis determines not only how the monad is used, but also how the monad is defined.

3.3 Termination of the evaluation function

An informal argument that `run_list` terminates for all finite input computations goes as follows. Let us order `Plus` computations by the number of effect-basis data constructors, break ties using the total number of data constructors, and break remaining ties using the left-nesting depth of `Bind` at the top level. The `run_list` function recursively evaluates only computations smaller than the original input because:

- Cases 1 and 6 are not recursive
- Case 4 eliminates two data constructors without introducing effect-basis data constructors
- Case 7 eliminates at least one effect-basis data constructor
- Case 5 reduces the top-level left-nesting depth of `Bind` without changing the composition of data constructors

Finally, inlining case 6 into case 2 (`Zero`) and case 7 into case 3 (`Plus`) shows the following property:

- Cases 2 and 3 lead to eliminating at least one effect-basis data constructor

Since the seven cases are exhaustive, `run_list` always returns a list when evaluating a finite `Plus` computation.

3.4 Compliance with monad laws

In this subsection I present a proof sketch that the evaluation transitions in `run_list` observationally enforce the monad laws by evaluating equivalent monadic computations to the same result. Let us consider a finite `Plus` computation evaluation context `C` as follows, where `m` represents a computation and `v` represents a value:

$$C ::= \bullet \mid \text{Bind } C (\backslash v \rightarrow m) \mid \text{Bind } m (\backslash v \rightarrow C) \mid \text{Plus } C \ m \mid \text{Plus } m \ C$$

In most of these contexts, the computation in the hole `•` does not affect the choice of the first evaluation transition. For example, the context `Plus C m` is always evaluated using case 7 regardless of what is in the hole `•` inside the context `C`. Thus, without loss of generality, we can let the evaluation proceed until the computation in the hole directly affects the choice of the evaluation transition. The context `C'` represents these cases:

$$C' ::= \bullet \mid \text{Bind } \bullet (\backslash v \rightarrow m)$$

The goal of the proof is to show that putting different sides of a monad law equivalence into the hole `•` in `C'` does not change the result of evaluation. Here I show the compliance with the right unit law by induction on the depth of the evaluation tree.

Evaluation context = `•`

$$\begin{aligned} & \text{run_list } (\text{Bind } (\text{Unit } v) \ \text{Unit}) \\ \rightarrow & \text{run_list } (\text{Unit } v) && \text{(case 4)} \\ & \text{run_list } \text{Zero} \\ \rightarrow & \text{run_list } (\text{Bind } \text{Zero } \text{Unit}) && \text{(case 2)} \\ & \text{run_list } (\text{Plus } m \ n) \\ \rightarrow & \text{run_list } (\text{Bind } (\text{Plus } m \ n) \ \text{Unit}) && \text{(case 3)} \\ & \text{run_list } (\text{Bind } (\text{Bind } m \ k) \ \text{Unit}) \\ \rightarrow & \text{run_list } (\text{Bind } m (\backslash v \rightarrow \text{Bind } (k \ v) \ \text{Unit})) && \text{(case 5)} \\ \rightarrow & \text{run_list } (\text{Bind } m (\backslash v \rightarrow k \ v)) && \text{(induction)} \\ \rightarrow & \text{run_list } (\text{Bind } m \ k) && (\eta \ \text{rule}) \end{aligned}$$

Evaluation context = `Bind • (\v → m)`

$$\begin{aligned} & \text{run_list } (\text{Bind } (\text{Bind } m \ \text{Unit}) \ g) \\ \rightarrow & \text{run_list } (\text{Bind } m (\backslash v \rightarrow \text{Bind } (\text{Unit } v) \ g)) && \text{(case 5)} \\ \rightarrow & \text{run_list } (\text{Bind } m (\backslash v \rightarrow g \ v)) && \text{(induction)} \\ \rightarrow & \text{run_list } (\text{Bind } m \ g) && (\eta \ \text{rule}) \end{aligned}$$

The derivations for the left unit and the associativity laws are quite similar and thus not produced here. Once we establish that `run_list` produces the same result for computations that can be proved equivalent in one step using the monad laws, we can use induction to extend the result to computations that can be proved equivalent in any finite number of steps.

Note that the term representation itself does not comply with the monad laws, and I focus only on observational equivalence. If two equivalent monadic computations always evaluate to the same result, it really should not matter if they do not have the same concrete representation.

3.5 Characteristics of operational monads

Defining monads with operations involves associating an evaluation transition with each top-level structure of monadic computations (as defined by the patterns in the evaluation function). This

formulation allows programmers to use the algebraic properties of the monad as evaluation transitions and to ensure the compliance with those properties by construction.

For example, in the `run_list` function, the transitions for cases 2–5 are from the monad laws, and the transition for case 7 uses the left distribution law of `MonadPlus`. Instead of proving that the denotation of a monad satisfies the desired algebraic properties, programmers adopting the operational approach can enforce the properties directly in the evaluation transitions. If the programmer uses the algebraic laws in all applicable cases, and the laws do not contradict each other, the monad defined through the evaluation should comply with all the desired laws.

Specifying evaluation in a case-by-case basis also achieves good separation of concerns. For example, in the `run_list` function, all behavior related to the monad laws is localized in cases 2–5, and all behavior related to the computational effects of the monad is localized in cases 1, 6, and 7. Such separation allows programmers to study certain aspects of the monad without reading through the entire evaluation function, and it also helps programmers to ensure that changing certain properties of the monad would not unexpectedly affect other properties.

Defining monads with operations is the basic idea behind the `Unimo` framework. In the next section I show how `Unimo` parameterizes both the term representation and the evaluation function to improve the modularity of operational monads.

4. Defining monads with Unimo

In this section I introduce the `Unimo` framework, which extracts the code common to all operational monads into a reusable form so that programmers need not create the same boilerplate code over and over again for each monad.

One objection to defining monads with operations is that an operational definition of a monad tends to be much more verbose than a denotational definition of the same monad. For example, the operational definition of the list monad in Figure 3 contains about three times as many lines of code as the denotational definition in Figure 1. However, careful inspection of Figure 3 shows that a significant part of the code is not specific to the list monad but needed by operational definitions of any monad. All monad term representations need the `Unit` and `Bind` data constructors, and all monad evaluation functions need transitions analogous to the cases 2–5 in `run_list` to enforce the monad laws.

The `Unimo` framework extracts the shared code into a parameterized core which implements the common behavior of a monad. Programmers define a `Unimo`-based monad by inserting the effect-basis data constructors into a parameterized term representation and by inserting the evaluation transitions for the effect basis into an observer function.

4.1 The Unimo term representation

The `Unimo` datatype in Figure 4 is the `Unimo` term representation which captures the syntactic structure of monadic computations. The `Unit` and `Bind` data constructors represent the monad operators, and the `Effect` data constructor represents a computation in the effect basis of the monad. The type parameter `r` in `Unimo`, which has kind $(* \rightarrow *) \rightarrow * \rightarrow *$, is the effect-basis datatype of the monad. For example, the `PlusE` datatype in Figure 5 represents the effect basis of the list monad. The type parameter `m` is the type of the term representation (in this case `Unimo PlusE`), and the type parameter `a` is the result type of the computation. The effect-basis data constructors are always used with the `Effect` constructor; for example, the `mzero` computation in the list monad which produces no results is represented as `Effect Zero`.

Each `Unimo` term representation is created by instantiating the `Unimo` datatype with a suitable effect-basis datatype. Since data

```

data Unimo r a
  = Unit a
  | Effect (r (Unimo r) a)
  | forall b. Bind (Unimo r b) (b → Unimo r a)
instance Monad (Unimo r) where
  return = Unit
  (≫=) = Bind
type BindOp r a v = forall b.
  r (Unimo r) b → (b → Unimo r a) → v
type Observer r a v =
  (a → v) → BindOp r a v → Unimo r a → v
observe_monad :: Observer r a v
observe_monad unit_op bind_op = eval where
  eval (Unit v)           = unit_op v
  eval (Effect e)        = e 'bind_op' Unit
  eval (Bind (Effect e) k) = e 'bind_op' k
  eval (Bind (Unit v) k)  = eval (k v)
  eval (Bind (Bind m k) g) = eval (Bind m cont)
  where cont v = Bind (k v) g

```

Figure 4. The Unimo monadic programming framework. The Unimo datatype is the term representation for all Unimo-based monads, and the observe_monad observer function implements the behavior common to all monads.

```

data PlusE m a
  = Zero
  | Plus (m a) (m a)
type Plus = Unimo PlusE
run_list :: Plus a → [a]
run_list = observe_monad unit_op bind_op where
  unit_op v           = [v]
  bind_op Zero _     = []
  bind_op (Plus m n) k =
    let ms = run_list (Bind m k)
        ns = run_list (Bind n k)
    in ms ++ ns

```

Figure 5. A Unimo-based definition of the list monad. The PlusE datatype declares the effect basis of the monad, and the run_list function invokes observe_monad with Unimo operators that implement the nondeterminism effect.

constructors in Unimo term representations are only placeholders for the actual monad operators and computations, they can be shared among multiple monads. For example, the Unit and Bind constructors are shared by all monads, and the Zero and Plus constructors are shared by monads in the MonadPlus type class (which also includes the Maybe monad). Because of this sharing, we can declare Unimo r as a trivial instance of the Monad class, and Unimo PlusE as a trivial instance of the MonadPlus class. These trivial instance declarations are useful only for compatibility with existing Haskell programs and for enabling the do-notation syntactic sugar. Since Unimo does not depend on overloading, it is just as easy to implement in a language without type classes or any other ad hoc polymorphism mechanisms as in Haskell.

4.2 The Unimo evaluation function

Similar to the Unimo term representation, the Unimo framework provides a parameterized *observer function* which defines the evaluation transitions common to all monads, and programmers can define an evaluation function of a monad by invoking the observer

```

data StateE s (m :: * → *) a where
  Get :: StateE s m s
  Put :: s → StateE s m ()
type State s = Unimo (StateE s)
run_state :: forall a s. State s a → s → (a, s)
run_state m s = observe_monad unit_op bind_op m where
  unit_op v           = (v, s)
  bind_op :: BindOp (StateE s) a (a, s)
  bind_op Get k       = run_state (k s) s
  bind_op (Put s1) k = run_state (k ()) s1

```

Figure 6. A Unimo-based definition of the state monad. The StateE datatype declares the effect basis of the monad, and the run_state function invokes observe_monad with Unimo operators that implement mutable state.

function with arguments that specify the transitions defining the desired computational effects.

The observer function observe_monad in Figure 4 is the core of all Unimo evaluation functions. The observe_monad function implements evaluation transitions analogous to the cases 2–5 of run_list in Figure 3, and it delegates the monad-specific cases — a lone return and an effect-basis computation bound to a continuation — to the *Unimo operators* unit_op and bind_op which observe_monad accepts as arguments. By extracting the common evaluation transitions into the observer function, Unimo eliminates the boilerplate in operational monads, and it provides a standard behavior of monads that programmers can plug in and ensure the compliance with monad laws automatically.

Figure 5 shows the run_list function adapted to the Unimo framework. The local function unit_op defines how to present the result of a list computation, and bind_op defines the semantics of the effect basis. Extracting all the shared transitions into the observer function makes the evaluation function shorter and allows the programmer to focus only on implementing the nondeterminism effect and not to be distracted by the monad laws.

4.3 Additional examples

In this subsection I present two more examples by implementing the state and the continuation monads in Unimo.

Figure 6 shows a Unimo-based definition of the state monad. Since the get and put computations produce results of specific types (the state type s and the empty tuple (), respectively), the effect-basis datatype StateE must be defined as a generalized algebraic datatype (GADT) [10] to associate its third type parameter with the data constructors. Definitions of the Unimo operators are straightforward: unit_op returns the computation result and the final state, and bind_op retrieves the current state for Get and overwrites the current state for Put.

Figure 7 shows a Unimo-based definition of the continuation monad. Even though the MonadCont type class consists of only a single computation callCC, here I extend the effect basis with a computation apply which uses its argument to replace the current computation. The Haskell continuation monad does not have an explicit apply because the continuation-passing style implicitly supports changing control flow at any point in the computation. What I do here is merely representing the implicit effect explicitly in the effect basis.

Defining an evaluation function for the continuation monad turns out to be easy because we can obtain the current continuation of a computation simply by taking the second argument of the ≫= operator. The CallCC case of bind_op builds a self-applying continuation cont using the current continuation k and runs the

```

data ContE r m a
  = forall b. CallCC ((a → m b) → m a)
  | Apply (m r)
type Cont r = Unimo (ContE r)
run_cont :: Cont a a → a
run_cont = observe_monad unit_op bind_op where
  unit_op v          = v
  bind_op (Apply m) _ = run_cont m
  bind_op (CallCC c) k = run_cont (Bind (c cont) k)
  where cont v = Effect (Apply (k v))

```

Figure 7. A Unimo-based definition of the continuation monad. The ContE datatype declares the effect basis of the monad, and the run_cont function invokes observe_monad with Unimo operators that capture and apply continuations.

computation c with cont as its argument. The Apply case gives up the current continuation and runs the computation m instead.

The three relatively short examples in this section demonstrate that Unimo is a general framework for defining a wide variety of monads with operations. Still, it is only through a complicated example that one can better appreciate the Unimo framework. I present one such example in the following section.

5. Parallel parsing processes

Parallel parsing processes [1] is a monad proposed by Claessen that eliminates the space leak exhibited by backtracking monadic parser combinator libraries [5]. When there are multiple parses for a sequence of symbols, a backtracking parser employs a depth-first search strategy that tries all parses in turn. When the monad pursues a successful parse, all alternative parses must be kept in memory until the current parse completes, thus creating a space leak. Parallel parsing processes solve the problem by pursuing all possible parses concurrently so that the unsuccessful parses can be quickly eliminated.

To develop the parallel parsing monad, Claessen started with a naïve term representation and applied a series of intricate program transformations to merge terms and to enforce the desired algebraic properties. (For example, one section in the paper was devoted to maintaining the associativity of the $\gg=$ operator.) In spite of the complicated reasoning employed in Claessen’s derivation, parallel parsing processes are based on only two simple ideas:

1. Classify a parse computation by its first effect-basis computation (if there is one), and then
2. Save the results of completed parses, discard failed parses, and continue processing the parses that require additional symbols.

The concept of a space leak, as well as the strategy of pursuing all parses concurrently, are both established deeply in the realm of operations, which makes Unimo the ideal framework for defining parallel parsing processes. In this section I present a Unimo-based implementation and explain its design in detail.

5.1 Defining the parser effect basis

Figure 8 shows the effect-basis datatype ParserE for a Unimo parser. A parser computation maps a sequence of symbols (type s) to a set of parses (type a). The effect basis of the monad has three elements: Symbol returns the next symbol in the sequence, Fail indicates a parse failure, and Choice poses two alternative parses for the same symbol sequence. The ParserE type defines a standard programming interface for monadic parser combinators; it is not in any way tailored toward parallel parsing processes. Whether

```

data ParserE s m a where
  Symbol :: ParserE s m s
  Fail   :: ParserE s m a
  Choice :: m a → m a → ParserE s m a
type Parser s = Unimo (ParserE s)

```

Figure 8. The effect-basis datatype of a parser monad. The definition of the effect basis in the ParserE datatype follows the formulation by Claessen [1].

```

type ParserOps s a = ([a], [s → Parser s a])
classify :: forall a s. [Parser s a] →
  ParserOps s a → ParserOps s a
classify []      ops          = ops
classify (p:ps) ops@(done, more) =
  observe_monad unit_op bind_op p where
  unit_op v          = classify ps (v:done, more)
  bind_op :: BindOp (ParserE s) a (ParserOps s a)
  bind_op Symbol k   = classify ps (done, k:more)
  bind_op Fail _     = classify ps ops
  bind_op (Choice u v) k = classify parsers ops
  where parsers = Bind u k : Bind v k : ps

```

Figure 9. The classification function of parallel parsing processes. The classify function defines a bind_op Unimo operator that classifies each parser computation based on its first effect-basis computation (or the lack thereof).

parsing is conducted depth-first or concurrently depends solely on the Unimo evaluation function.

5.2 Classifying parse computations

Figure 9 shows the classify function which classifies a list of parse computations by their structures. The purpose of the classification is to plug space leaks by identifying failed parses early without fully evaluating the successful parses. The classify function implements this functionality by defining a Unimo operator bind_op that performs a case analysis on the computations. Parses that need more symbols are collected in the more variable without further evaluation, and failed parses are discarded directly. If a parse starts with a Choice constructor, bind_op expands the embedded parses and classifies them using the same rules. Finally, the unit_op operator collects completed parses in the done variable.

Here we see the power of capturing a monadic computation as algebraic data and processing it with a function. Coding a similar classification procedure into the denotation of a monad, as done by Claessen, requires more convoluted reasoning and produces code that is more difficult to understand.

5.3 Evaluating classified parse computations

Figure 10 shows the evaluation function parallel_parse which pursues all possible parses concurrently. The function takes a list of parse computations (parsers) and a list of symbols (symbols) as input, and for each successful parse it returns a pair containing the parse result and the remaining symbols that are not consumed in the parse.

The parallel_parse function is defined recursively; one symbol is consumed in each invocation until all parses have terminated or all symbols have been consumed. The results variable contains the results of successful parses that terminate in this invocation, and the further function evaluates the parses that are still in progress by supplying one additional symbol with feed and then invoking the parallel_parse function.

```

parallel_parse :: [Parser s a] → [s] → [(a, [s])]
parallel_parse [] _ = []
parallel_parse parsers symbols =
  let (done, more) = classify parsers ([], [])
      results      = zip done (repeat symbols)
      further []   = []
      further (s:ss) = parallel_parse (feed s) ss
      feed s       = map ($) s more
  in results ++ further symbols

```

Figure 10. The evaluation function of parallel parsing processes. The `parallel_parse` function collects the results of completed parses and feeds one additional symbol to the computations that are still in progress.

5.4 Comparison with Claessen’s implementation

The Unimo-based implementation of parallel parsing processes presented in this section is significantly simpler than the original implementation by Claessen [1]. While Claessen’s implementation is a little shorter, coming up with the `SymbolBind` and `ReturnPlus` constructors and applying context passing [4] to enforce the right-associativity of $\gg=$ require ingenious thinking, careful derivation, and an intimate knowledge of advanced functional programming techniques. In contrast, the Unimo-based implementation consists of two recursive functions on lists, which is a technique already well-understood by most functional programmers.

The modular designs of both the Unimo term representation and the evaluation function greatly reduce the boilerplate required to define a monad. As a result, programmers can concentrate on the strategy for parallel parsing with little distraction. The evaluation transitions in the observer function not only ensures that the implementation complies with monad laws, but also flattens a parse computation into a sequence of effect-basis computations and greatly simplifies the definition of the classification function.

I have translated the grammar of the Oberon programming language [19] into a monadic parser and benchmarked the two parallel parser monads using sample code from Project Oberon [20]. Compared to Claessen’s implementation, the Unimo-based implementation is 20%–40% slower but consumes roughly the same amount of memory.

6. Defining monad transformers with Unimo

A *monad transformer* [14] is a mechanism for adding computational effects onto an existing monad (which is usually called the *underlying monad*). For example, a state monad transformer adds mutable state to a monad. In Haskell a monad transformer is a type constructor parameterized by the type constructor of the underlying monad. In addition to the monad operators and an effect basis, a monad transformer also has a lift operator which lifts a computation in the underlying monad into the transformed monad. The lift operator allows programmers to access the computational effects of the underlying monad in the transformed monad.

Monad transformers are important not only because they allow programmers to build monads with multiple effects in a modular manner, but also because they are a theoretical framework for studying the interaction of computational effects. In this section I extend Unimo to support monad transformers.

6.1 Monad transformers in Haskell

Figure 11 shows a list monad transformer in Haskell which adds nondeterminism support to the underlying monad. The code is from the Haskell Hierarchical Libraries with a little refactoring for clearer presentation. In principle, a monad transformer is like the corresponding monad, except that it has to preserve effects in the

```

newtype ListT m a =
  ListT { runListT :: m [a] }

instance Monad m => Monad (ListT m) where
  return x = ListT (return [x])
  m >>= k = ListT $
    do xs <- runListT m
       concatMapM (runListT . k) xs

concatMapM :: Monad m => (a → m [b]) → [a] → m [b]
concatMapM f = loop where
  loop [] = return []
  loop (x:xs) = do us <- f x
                  vs <- loop xs
                  return (us ++ vs)

instance MonadTrans ListT where
  lift m = ListT $ do a <- m
                    return [a]

instance Monad m => MonadPlus (ListT m) where
  mzero = ListT (return [])
  mplus u v = ListT $
    do us <- runListT u
       vs <- runListT v
       return (us ++ vs)

```

Figure 11. The list monad transformer in Haskell. The definition here is based on the same principle as the list monad in Figure 1, but interaction with the underlying monad makes the code much more complicated.

underlying monad by sequencing computations with the underlying $\gg=$ operator. In practice, the adaptation requires extensive and intrusive changes, and the list monad transformer in Figure 11 no longer resembles the list monad shown in Figure 1. Here I detail the main changes:

1. We need to extend the denotation of list monad computations to incorporate the underlying monad. We define a monad type constructor `ListT` which denotes a transformed computation as an underlying computation that produces a list of results.
2. We need to update the monad operators to match the new denotation of computations. The $\gg=$ operator becomes much more complicated because applying the list of results `xs` to the continuation `k` produces multiple sub-computations which all need to be sequenced with the underlying $\gg=$ operator.
3. We need to define a lift operator and make `ListT` an instance of the `MonadTrans` monad transformer type class.
4. Finally, we need to update the effect basis to match the new denotation of computations. In particular, `mplus` now needs to sequence its parameters `u` and `v` because they are not lists but (conceptually) computations that produce lists.

That is a lot of work, and many changes require a certain amount of ingenuity. To make matters worse, the list-transformed monad does not satisfy the associativity law if the underlying monad is not commutative [14]. Consider the following example where `a`, `b`, `c`, `d` are lifted IO computations that print the corresponding characters:

```

t1 :: ListT IO ()
t1 = (mplus a b >> c) >> d

t2 :: ListT IO ()
t2 = mplus a b >> (c >> d)

```

By the associativity law, `t1` and `t2` should produce the same results, but evaluating `t1` prints `abccdd`, and evaluating `t2` prints `abcdcd`.

Hinze proposed a backtracking monad transformer that does not depend on a commutative underlying monad [3], but his technique requires a completely different design strategy, and to the best of my knowledge there is no simple way to fix the code in Figure 11 to eliminate the problem.

6.2 The principles of monad transformers

There have been several prior works on the principles of monad transformers. Liang and associates proposed the following *monad transformer laws* [14]:

$$\begin{aligned} \text{lift} \cdot \text{return}_u &\equiv \text{return}_t \\ \text{lift} (m \gg=_{=u} k) &\equiv (\text{lift } m) \gg=_{=t} (\text{lift} \cdot k) \end{aligned}$$

The suffix of each monad operator denotes whether it is in the transformed monad (t) or in the underlying monad (u). The laws are of little value to the designers of monad transformers because they say nothing about important issues such as the integration of the underlying monad into the monad transformer. Later, Hinze characterized a monad transformer as a type constructor τ with a pair of functions `promote` and `observe` [3]:

$$\begin{aligned} \text{promote} &:: \text{Monad } m \Rightarrow m \ a \rightarrow \tau \ m \ a \\ \text{observe} &:: \text{Monad } m \Rightarrow \tau \ m \ a \rightarrow m \ a \end{aligned}$$

The `promote` function is the `lift` operator in Haskell, and the `observe` function maps a computation in the transformed monad back to the underlying monad by implementing the computational effects introduced by the monad transformer. The `promote` function must satisfy the monad transformer laws, and in addition the two functions must also satisfy the following laws:

$$\begin{aligned} \text{observe} \cdot \text{return}_t &\equiv \text{return}_u \\ \text{observe} (\text{promote } m \gg=_{=t} k) &\equiv m \gg=_{=u} (\text{observe} \cdot k) \end{aligned}$$

The last law is particularly useful for designing monad transformers because it points out that a monad transformer needs to process only the effect basis defined by the transformer and can pass lifted computations as part of the output.

Even though Hinze’s characterization provides us with a good understanding of how monad transformers work, the characterization does not generalize beyond the Haskell backtracking monad transformer. The problem lies in the return type of `observe`. For example, `observe` for the list monad transformer should have the following type:

$$\text{observe} :: \text{Monad } m \Rightarrow \text{ListT } m \ a \rightarrow m \ [a]$$

The state monad transformer `StateT`, which adds support for mutable state, has an `observe` function with the following type:

$$\text{observe} :: \text{Monad } m \Rightarrow \text{StateT } s \ m \ a \rightarrow s \rightarrow m \ (a, s)$$

The general `observe` type listed previously cannot describe either of these examples. Hinze did not run into this problem because his backtracking monad returns only one result and the `Monad` type class in Haskell happens to include a fail computation. However, if we wish to use this characterization as a basis for Unimo-based monad transformers, the formulation of `observe` must be generalized to cover these cases as well. I will show how to accomplish this task in the following subsections.

6.3 Extending Unimo to monad transformers

Figure 12 shows an extended version of the Unimo framework that supports monad transformers. The definitions here replace those in Figure 4, and I grayed out the parts that remain the same to highlight the changes. The `UnimoT` datatype includes a `Lift` data constructor to represent the lift operator, and the `Unimo` datatype

```

data UnimoT r m a
= Unit a
| Lift (m a)
| Effect (r (UnimoT r m) a)
| forall b. Bind (UnimoT r m b) (b → UnimoT r m a)

instance Monad (UnimoT r m) where
  return = Unit
  (>>=) = Bind

instance MonadTrans (UnimoT r) where
  lift = Lift

data Xm a
type Unimo r = UnimoT r Xm

type BindOpT r m a v = forall b.
  r (UnimoT r m) b → (b → UnimoT r m a) → v
type BindOp r a v = BindOpT r Xm a v

type BindU m v = forall b. m b → (b → v) → v
type Observer r m a v =
  (a → v) → BindOpT r m a v → UnimoT r m a → v

observe_core :: BindU m v → Observer r m a v
observe_core bind_u unit_op bind_op = eval where
  eval (Unit v)           = unit_op v
  eval (Effect e)         = e 'bind_op' Unit
  eval (Bind (Effect e) k) = e 'bind_op' k
  eval (Lift m)           = m 'bind_u' unit_op
  eval (Bind (Lift m) k)  = m 'bind_u' (eval . k)
  eval (Bind (Unit v) k)  = eval (k v)
  eval (Bind (Bind m k) g) = eval (Bind m cont)
  where cont v = Bind (k v) g

observe_monad :: Observer r Xm a v
observe_monad = observe_core undefined

observe_trans :: Monad m ⇒ Observer r m a (m v)
observe_trans = observe_core (>>=)

```

Figure 12. The Unimo programming framework extended to support Unimo transformers. The definitions here supersede those in Figure 4, and I grayed out the common code fragments to highlight the changes.

becomes a special case of `UnimoT` in which the underlying monad is instantiated with the empty type `Xm` to prevent `Lift` from being used. To deal with the `Lift` data constructor, the observer function `observe_core` adds two evaluation transitions using the right unit law and Hinze’s second law of `observe`.

Note that in addition to the Unimo operators, `observe_core` is also parameterized by the `>>=` operator in the underlying monad. This parameterization makes `observe_core` backwards compatible with `observe_monad`. Substituting `>>=` for `bind_under` requires the result of `observe_core` to be a computation in the underlying monad, but this formulation is incompatible with `observe_monad` because Unimo uses `Xm` as a dummy underlying monad. The `bind_under` parameter allows `observe_monad` and `observe_trans` to share the same code base and therefore reduces duplication in the framework. Programmers should use either `observe_monad` (for Unimo) or `observe_trans` (for Unimo transformer) instead of invoking `observe_core` directly.

6.4 Unimo-based monad transformers

The simplest way to build a Unimo-based monad transformer is to adapt a Unimo-based monad through the following procedure.

```

type PlusT = UnimoT PlusE
run_list_t :: Monad m => PlusT m a -> m [a]
run_list_t = observe_trans unit_op bind_op where
  unit_op v = return [v]
  bind_op Zero _ = return []
  bind_op (Plus m n) k =
    do ms <- run_list_t (Bind m k)
       ns <- run_list_t (Bind n k)
    return (ms ++ ns)

```

Figure 13. A Unimo-based definition of the list monad transformer. The code is based on Figure 5 with the common fragments grayed out. Compared to the standard definition in Figure 11, the Unimo-based definition is not only shorter but also works with a non-commutative underlying monad.

```

type StateT s = UnimoT (StateE s)
run_state_t :: forall a s m. Monad m =>
  StateT s m a -> s -> m (a, s)
run_state_t m s = observe_trans unit_op bind_op m where
  unit_op v = return (v, s)
  bind_op :: BindOpT (StateE s) m a (m (a, s))
  bind_op Get k = run_state_t (k s) s
  bind_op (Put s1) k = run_state_t (k ()) s1

```

Figure 14. A Unimo-based definition of the state monad transformer. The code is based on Figure 6 with the common fragments grayed out.

1. Rename the term representation and the evaluation function to avoid conflict.
2. Use the Unimo definitions that support the lift operator. The change requires replacing Unimo with UnimoT, BindOp with BindOpT, and observe_monad with observe_trans.
3. Make the evaluation function produce computations in the underlying monad as result. The change requires modifying type signatures and the Unimo operators. An evaluation transition without recursive evaluations needs a return, and a transition with recursive evaluations may need to use the $\gg=$ operator to sequence the underlying computations.

Figure 13 shows a Unimo-based list monad transformer adapted from the Unimo-based list monad in Figure 5. The only nontrivial change is that `bind_op` now sequences the sub-computations built from `m` and `n` in the `Plus` transition, and this definition of the list monad transformer does not require a commutative underlying monad (both computations `t1` and `t2` in Section 6.1 print `acdbcd`). Figure 14 shows a Unimo-based state monad transformer adapted from the Unimo-based state monad in Figure 6. All of the changes are pretty trivial, and the evaluation transitions for the effect basis are pretty much left as they were in Figure 6.

Though I have not done any in-depth analysis on this procedure, it appears to be applicable to a wide variety of Unimo-based monads, including the continuation monad in Section 4.3 and the parallel parsing monad in Section 5. By providing a common infrastructure for both monads and monad transformers, the Unimo framework conceals their superficial differences and exposes the underlying connections between these two constructs.

7. Meta-programming with Unimo

A monadic computation in Haskell is typically a black box due to the use of functions to define monad operators and the effect basis.

```

observe_xlate :: Observer p m u (UnimoT q m v)
observe_xlate = observe_core (Bind . Lift)

```

Figure 15. The observer function for Unimo translators. The code references definitions in Figures 12.

While a program can run the computation and obtain some kind of result, nothing else can be done with the computation. Such a limitation does not exist in the Unimo framework: since the monad operators and effect-basis computations are data constructors, the programmer can open up a monadic computation and see what happens inside. In this section I show how to use this capability to support meta-programming on Unimo computations.

7.1 Observer function for Unimo translators

Meta-programming in the Unimo framework is built on top of *Unimo translators*. A Unimo translator is like an evaluation function, but instead of evaluating the computation, the translator generates another computation as its result. Using a translator, a programmer can take a Unimo computation apart, make some changes, and put everything back together.

Figure 15 shows the observer function for Unimo translators. The observer function `observe_xlate` is also based on `observe_core`, but it handles a computation in the underlying monad by lifting it back to the transformed monad so that it can be bound back to its continuation. To define a Unimo translator, the programmer defines Unimo operators that translate the effect basis and combines the operators with `observe_xlate`.

Note that the translation takes place not when an effect-basis computation is defined but when the effect-basis computation is executed. If a single effect-basis computation is executed multiple times (for example, due to recursion), it will be translated many times with possibly different results. This behavior is unavoidable because the translator works on a runtime data value instead of the Haskell source code. Runtime translation incurs a heavier performance penalty, but it is also more general because the translation process can make use of runtime information.

In the remainder of this section I will provide two examples of Unimo meta-programming on tracing and sandboxing Unimo computations.

7.2 Tracing Unimo computations

Direct observation is a powerful technique for understanding the behavior of a system. When a monadic computation behaves in an unexpected manner, the first step of debugging is to find out what exactly is going on under the hood. While some monads support output, the programmer still needs to insert trace code into the computation to perform the output operations. Manual insertion of trace code works for simple cases, but it is much better to have a program do it while the computation executes so that there is no need to modify the source code of the computation.

Figure 16 shows the `trace_monad` function which produces a self-tracing Unimo computation. The traced computation uses the `note` function to maintain a summary `s` of its execution. When the evaluation reaches an effect-basis computation, it invokes `note` to update the summary based on the executed computation and its result. When the computation reaches the end, it returns a pair consisting of the result and the final execution summary. The `trace_monad` function is designed with flexibility in mind: by using an appropriate `note` function, the programmer can produce the complete execution history, count how many times an effect-basis computation is executed, or compute the distribution of the arguments of effect-basis computations.

Even though `trace_monad` may look reasonable, there are still cases where it fails to work as expected. The problem lies in the

```

type Note r m o = forall b. r (UnimoT r m) b → b → o
trace_monad :: forall r m a s. Note r m (s → s) → s →
  UnimoT r m a → UnimoT r m (a, s)
trace_monad note = run_trace where
  run_trace s = observe_xlate unit_op bind_op where
    unit_op v = Unit (v, s)
    bind_op :: BindOpT r m a (UnimoT r m (a, s))
    bind_op e k = do v ← Effect e
      run_trace (note e v s) (k v)

```

Figure 16. A Unimo translator that implements transparent tracing of monadic computations. The `trace_monad` function maintains an execution summary through the `note` function and returns the summary when the computation terminates normally.

```

trace_trans :: forall r m a. Monad m ⇒ Note r m (m ()) →
  UnimoT r m a → UnimoT r m a
trace_trans note = run_trace where
  run_trace = observe_xlate unit_op bind_op
  unit_op v = Unit v
  bind_op :: BindOpT r m a (UnimoT r m a)
  bind_op e k = do v ← Effect e
    Lift (note e v)
    run_trace (k v)

```

Figure 17. A Unimo translator that implements transparent tracing of monad transformer computations. Instead of manually maintaining an execution summary, `trace_trans` implements tracing by inserting computations in the underlying monad.

decision to pass the execution summary as part of the result of the computation: if the traced computation produces no results — which is possible in the list monad — it will also fail to produce a summary. Figure 17 shows the `trace_trans` function which uses computations in the underlying monad as trace code. While `trace_trans` can be used only with monad transformers, it does solve the aforementioned problem with `trace_monad`. If the programmer uses `Writer` or `IO` as the underlying monad, the traces are guaranteed to be available no matter what happens in the transformed monad.

Neither of these general trace functions is completely satisfactory. Nonetheless, I hope that they illustrate the basic design principles of tracing Unimo computations that programmers can use when designing their own application-specific trace functions.

7.3 Sandboxing Unimo computations

Another use of the meta-programming feature in Unimo is sandboxing an IO computation to make sure that it does not perform any unauthorized operations. Haskell uses the IO monad to encapsulate a wide variety of imperative effects, including input–output, operating system calls, foreign-function invocations, asynchronous exceptions, and concurrency. Such a design makes it impossible to separate different effects, and an IO computation used to catch an asynchronous exception is totally indistinguishable from another that deletes all files on the hard drive. In this subsection I show how to execute an IO computation in a sandbox that disables certain features provided by the monad.

Figure 18 shows a miniature version of the effect-basis datatype of the Unimo-based IO monad. In contrast with the standard formulation of IO in Haskell, the Unimo-based definition does not require IO to be an abstract datatype. As long as the language implementation does not provide an evaluation function for IO, it is perfectly

```

data IOE m a where
  OpenFile :: FilePath → IOMode → IOE m Handle
  Close    :: Handle → IOE m ()
  FileSize :: Handle → IOE m Int
  GetChar  :: Handle → IOE m Char
  PutChar  :: Handle → Char → IOE m ()
  IOError  :: IOError → IOE m a
  IOCatch  :: m a → (IOError → m a) → IOE m a
type IO = Unimo IOE

```

Figure 18. The effect-basis datatype of a small subset of the Haskell IO monad. In the Unimo framework, there is no need to define IO as an abstract datatype because it is only a term representation. The “magic” of IO lies entirely in its evaluation function which is generally not available to programmers.

```

deny :: IO a → IO a
deny = observe_xlate unit_op bind_op where
  unit_op v = Unit v
  bind_op e k = Bind (Effect (xlate e)) (deny . k)
  xlate (OpenFile _ _) = IOError (userError message)
  xlate (IOCatch m h) = IOCatch (deny m) (deny . h)
  xlate e = e
  message = "file open denied"

```

Figure 19. A Unimo translator that implements sandboxing. The `deny` function traverses through the computation and replaces all occurrences of `OpenFile` with `IOError`.

safe to allow programmers to manipulate the IO term representation like any other data structure.

Figure 19 shows the `deny` function which verifies at runtime that an IO computation does not open any files. The sandboxing is achieved by mapping the `OpenFile` constructor to `IOError` before execution and leaving all other constructors unchanged. Note that the `xlate` function includes a case for the `IOCatch` constructor: even though `IOCatch` does not open any files, it contains two embedded computations that also need to be sandboxed. Without this special treatment, a malicious programmer can avoid detection by opening a file in an `IOCatch` computation.

Even though I use IO as the example, the same technique can also be used to implement assertions on monadic computations to facilitate early discovery of programming errors.

8. Related work

The Unimo framework has interesting connections to many old and new ideas in programming languages. Here I discuss some of the most significant ones and explain how they relate to this work.

8.1 Representing code as data

The design of Unimo is heavily influenced by the idea of representing code as data that originated in Lisp [11]. Using data constructors as placeholders for monad operators and the effect basis, Unimo turns computations into algebraic data that are later transformed and evaluated.

Nonetheless, there are significant differences between the Lisp `eval` function and a Unimo evaluation function. While the former accepts as input an S-expression that encodes the abstract syntax of a Lisp program, the latter processes the monadic computations produced by evaluating Haskell expressions. Therefore, instead of saying that Unimo captures code as data, a more accurate description would be that Unimo captures *computations* as data. Working with computations instead of raw source code limits the expres-

siveness of Unimo evaluation functions, but on the other hand, it also offloads most of the work to the Haskell implementation and simplifies the design of Unimo considerably.

8.2 Deriving monads from term representations

The derivation technique proposed by Hughes [4] and later used by Hinze [3] and Claessen [1] illustrates how programmers can derive a monad from its intended semantics. Unimo follows the same basic strategy of using term representations to capture the structure of computations, but the two techniques differ significantly in other aspects.

By using generalized algebraic datatypes [10], Unimo provides a modular term representation that is shared by all monads. Having a standard term representation makes it possible to define an observer function that implements the standard semantics of all monads. Instead of presenting a methodology that illustrates how to build a monad from scratch, Unimo provides working code that programmers can use for defining monads. Unimo should allow programmers to enjoy some benefits of deriving monads without going through the derivation process and building an implementation from the ground up.

8.3 Reactive resumption monads

Harrison and Hook investigated using reactive resumption monads to model computations that interact with the environment [2]. A reactive resumption monad differs from a resumption monad in that the computation makes a request of type q when it pauses, and the environment must provide a response of type r to resume the computation. Stated simply, reactive resumption allows a computation to request a service from the environment and wait until a response is available before continuing execution.

A reactive resumption monad is similar to a Unimo-based monad: the constructors of q correspond to the effect basis which produces results of type r . Using generalized algebraic datatypes [10], Unimo allows effect-basis computations to produce results of different types and thus enables programmers to build more faithful representations of monadic computations.

8.4 Aspect-oriented programming

Aspect-oriented programming is a paradigm which aims to address the issue of cross-cutting concerns that cannot be extracted with typical modularity mechanisms [13]. In a system like AspectJ [12], an aspect consists of pointcut declarations which specify certain points of program execution, and advice declarations which specify when the inserted code should execute in relation to the defined pointcuts. Using aspects, a programmer can specify, for example, that the moved field of an object be set to true after the x field of the object is set.

De Meuter proposed using monads as a theoretical foundation for aspect-oriented programming [15]. In his design, weaving an aspect into a computation requires redefining the monad operators, which makes the use of aspects much more complicated than in systems like AspectJ. Also, since pointcuts are identified by the implementation of the $\gg=$ operator, the programmer must tag every use of $\gg=$ with the first effect-basis computation in the right-hand-side continuation. This requirement makes the design impractical for computations that involve dynamic control flow.

We can characterize the Unimo translators described in Section 7 as aspects on monadic computations by considering each effect-basis computation as a potential pointcut. For example, the deny translator in Figure 19 can be thought of as an aspect that replaces every call to `OpenFile` with a call to `IOError`. Monad translators are more powerful than aspects in AspectJ because the programmer can define pointcuts with arbitrary computations, and in addition to inserting code, a monad translator can also alter an effect-basis com-

putation identified in the pointcut. Even though monad translators can change a computation in complicated ways, the Haskell type system ensures that all changes are type correct and allows programmers to work safe in the knowledge that the translators will always generate well-formed computations.

9. Conclusions and future work

The development of the Unimo framework demonstrates that capturing monadic computations as algebraic data is a powerful idea. Using a term representation makes it unnecessary to assign meanings to individual monad operators and effect-basis computations; instead programmers define an evaluation function that assigns a meaning to a monadic computation as a whole. Such an integrated approach to semantics frees programmers from having to unravel the tangled web of operator interactions through analytic means, and it allows them to realize the interactions directly as evaluation transitions which are executed at runtime. The straightforward definition of the list monad transformer in Section 6.4 should serve as a testament to the power of this approach.

Another indispensable ingredient of the success of Unimo is modularity. The `UnimoT` datatype, used in all Unimo term representations, serves as the foundation of all monads and monad transformers. Without `UnimoT`, it would be difficult to define the generic `observe_core` observer function, which in turn plays an important role in discovering the correspondence between monads and monad transformers discussed in Section 6. The `UnimoT` datatype relies on generalized algebraic datatypes, which reinforces the idea that expressive type systems can lead to the discovery of new programming techniques.

One difficulty I encountered when developing Unimo-based monads was the need to supply detailed type annotations; sometimes it takes much more effort to convince the type checker that a function is type safe than to write the (type safe) function itself. Figure 6 shows one such example. The `run_state` function makes the states s visible to the Unimo operators through lexical scoping. The function is type correct, but it takes two type signatures with a lexically scoped type variable to convince GHC that that is indeed the case. If we can identify common programming patterns involving existentially quantified type variables and GADTs and incorporate such knowledge directly into the Haskell type checker, I believe that Unimo programming would become an even more enjoyable experience.

This paper covered quite a bit of ground, but more work remains to be done. In Section 4.2 I claimed that an evaluation function can guarantee compliance with monad laws by using the observer function to evaluate the term representations. While this statement appears to be obvious from the definition of the observer function, extending the proof sketch for the operational list monad in Section 3.4 to Unimo remains an important goal.

In Section 6 I outlined a procedure for adapting a Unimo evaluation function to support Unimo transformers. Even though the procedure works in a lot of cases, the extent to which it is applicable remains unknown. A formal characterization of the adaption procedure should not only be of great practical importance, but also provide further insight into the relationship between monads and monad transformers.

Finally, in Section 8.4 I described Unimo translators as a generalization of aspects on monadic computations. It would be interesting to see how useful the technique turns out to be in addressing cross-cutting concerns in monadic computations, and whether we can package the translators in a way that is accessible to most Haskell programmers.

Acknowledgments

The example in Section 6.1 that demonstrates the non-associative nature of the naïve list monad transformer is attributed to Roberto Zunino on the Haskell wiki website.

This work is partially supported by the National Science Foundation of the United States under grants CCR-0219686 and CNS-0523474. I would like to thank Mark P. Jones, Jim Hook, Tim Sheard, Tom Harke, Iavor S. Diachki, Thomas Hallgren, Andrew Tolmach, and Dan Brown for their comments on this work and feedbacks on drafts of this paper.

References

- [1] Koen Claessen. Functional Pearl: Parallel parsing processes. *Journal of Functional Programming*, 14(6):741–757, November 2004.
- [2] William Harrison and James Hook. Achieving information flow security through precise control of effects. In *18th IEEE Computer Security Foundations Workshop*, pages 16–30, Los Alamitos, CA, USA, June 2005. IEEE Computer Society.
- [3] Ralf Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 186–197, New York, NY, USA, September 2000. ACM Press.
- [4] John Hughes. The design of a pretty-printing library. In Johan Jeuring and Eric Meijer, editors, *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *LNCS*, pages 53–96. Springer, May 1995.
- [5] John Hughes. Generalizing monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
- [6] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In CAR Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering Theories of Software Construction*, volume 180 of *NATO Science Series: Computer & Systems Sciences*, pages 47–96. IOS Press, 2001.
- [7] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, May 2003.
- [8] Simon Peyton Jones and Mark Shields. Lexically scoped type variables. Submitted to ICFP 2004, March 2004.
- [9] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. Unpublished manuscript, July 2005.
- [10] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalized algebraic data types. Technical Report MS-CIS-05-26, University of Pennsylvania, July 2004.
- [11] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. *ACM SIGPLAN Notices*, 28(3):231–270, March 1993.
- [12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming: 15th European Conference*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, June 2001.
- [13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, June 1997.
- [14] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, New York, NY, USA, January 1995. ACM Press.
- [15] Wolfgang De Meuter. Monads as a theoretical foundation for AOP. Position paper in ECOOP’97 Workshop on Aspect-Oriented Programming, June 1997.
- [16] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, June 1989. IEEE Press.
- [17] Eugenio Moggi. Notions of computation and monads. *Information And Computation*, 93(1):55–92, July 1991.
- [18] Peter D. Mosses. Denotational semantics. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 11, pages 575–631. MIT Press, December 1990.
- [19] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula*. Addison Wesley, Reading, MA, USA, June 1992.
- [20] Niklaus Wirth. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, Reading, MA, USA, November 1992.